AD-A282 930

MENTATION PAGE

Form Approved
OMB No. 0704-0188

2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
8 June 1994	FINAL-9/15/91 to 9/14/93

4. TITLE AND SUBTITLE
Microcomputer-based Aircraft Routing and Scheduling

5. FUNDING NUMBERS

DTIC ELECTE AUG 03 1994 S B D

6. AUTHOR(S)
Mr. Dan Greenwood
Dr. Kendall Nygard

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NETRLOGIC Inc., 5080 Shoreham Place, St 201
San Diego, CA 92122-5932
North Dakota State University, Dept. of Comp. Sci. and Oper. Res., Fargo, ND 58105-5075

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 94 0422

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR NM
110 DUNCAN AVE SUITE B115
BOLLING AFB DC 20332-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-91-C-0078

11. SUPPLEMENTARY NOTES

94-24525

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
Distribution unlimited

12b. DISTRIBUTION CODE
UL

13. ABSTRACT (Maximum 200 words)

The work concerns research, development and software implementation of distributed mathematical optimization algorithms for aircraft routing and scheduling. Two Air Force application concerns customized short-term scheduling of military aircraft to support mission-critical passenger travel. The mission routing and scheduling procedures developed under the project were coded, and a graphical user interface was developed for prototype use by United States Air Forces in Europe at Ramstein AFB in Germany.

In addition to the optimization procedures and the graphical user interface, a specialized underlying database management system was developed from scratch, to support an airport atlas and information pertaining to aircraft characteristics, traveling personnel, and other relevant information. The system runs UNIX workstation computers under the X-Window environment with Motif. The optimization model and distributed computing model were designed and developed to facilitate expansion into use for military passenger fleet units that are highly distributed geographically, setting the stage for strong coordination of airlift activities among multiple sites and different branches of the armed forces. The optimization approach is based on set partitioning, and uses global methods inspired by price-directive decomposition. Recent advances in artificial intelligence for distributed problem solving, and recent capabilities in wide-area communication technology also play a role.

14. SUBJECT TERMS	15. NUMBER OF PAGES
94 8 02 213	162
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Phase II Final Report

Microcomputer-based Aircraft Routing and Scheduling

AFOSR SBIR 90-190

Contract Nr. F49620-91-C-0078

Submitted to:

Dr. Neal Glassman
Directorate of Mathematical and Information Sciences
Air Force Office of Scientific Research
Bolling AFB
Washington D. C.

Submitted from:

Mr. Dan Greenwood
Netrologic Incorporated
5080 Shoreham Place, Ste 201
San Diego, Ca 92122
(619)-458-1624

Dr. Kendall E. Nygard
Department of Computer Science and Operations Research
North Dakota State University
Fargo, ND 58105-5075
(701)-237-8203, (701)-777-3870 and (701)-777-3179
nygard@plains.nodak.edu

TABLE OF CONTENTS

Abstract

The work concerns research, development and software implementation of distributed mathematical optimization algorithms for aircraft routing and scheduling. The Air Force application concerns customized short-term scheduling of military aircraft to support mission-critical passenger travel. The mission routing and scheduling procedures developed under the project were coded, and a graphical user interface (GUI) was developed for prototype use by United States Air Forces in Europe (USAFE) at Ramstein AFB in Germany.

In addition to the optimization procedures and the GUI, a specialized underlying database management system was developed from scratch, to support an airport atlas and information pertaining to aircraft characteristics, traveling personnel, and other relevant information. The system runs UNIX workstation computers under the X-Window environment with Motif. The optimization model and distributed computing model were designed and developed to facilitate expansion into use for military passenger fleet units that are highly distributed geographically, setting the stage for strong coordination of airlift activities among multiple sites and different branches of the armed forces. With modern telecommunications capabilities, the internet or other wide-area networks would be used as the communication mechanism. Local autonomy and control would be retained, yet support globally optimal airlift scheduling. The optimization approach is based on set partitioning, and uses global methods inspired by price-directive decomposition. Recent advances in artificial intelligence for distributed problem solving, and recent capabilities in wide-area communication technology also play a role.

The report is organized into three major Sections. Section 1 provides technical descriptions of the research that has been accomplished, including the client/server distributed computing model, the extended set partitioning optimization model, and the ways in which the system can be used to advantage. Section 2 is a user manual for the Dakota software system. Schedulers who wish to actually use the system in practice to schedule aircraft will make extensive use of the material in this section. There are descriptions of the various software environments and how to use them, and also a description of the underlying database system. Section 3 provides an abstract description of the database management system that was developed for the project. This material would be of basic interest to anyone who wishes to significantly expand or modify the system.

Introduction

The work concerns research, development and software implementation of distributed mathematical optimization algorithms for aircraft routing and scheduling. Two Air Force applications were the driving forces for the project. First, during the initial months of the project, the investigations involved methods that would apply to the Air Force LOGAIR cargo transport system in the continental United States. However, the LOGAIR system was discontinued while the work was underway, and a decision was made to direct the effort to another Air Force application, that of customized short-term scheduling of military aircraft to support mission-critical passenger travel. The mission routing and scheduling procedures developed under the project were coded, and a graphical user interface (GUI) was developed for prototype use by United States Air Forces in Europe (USAFE) at Ramstein AFB in Germany. In addition to the optimization procedures and the GUI, a specialized underlying database management system was developed from scratch, to support an airport atlas and information pertaining to aircraft characteristics, traveling personnel, and other relevant information. The system runs UNIX workstation computers under the X-Window environment with Motif. The optimization model and distributed computing model were designed and developed to facilitate expansion into use for military passenger fleet units that are highly distributed geographically, setting the stage for strong coordination of airlift activities among multiple sites and different branches of the armed forces. With modern telecommunications capabilities, the internet or other wide-area networks would be used as the communication mechanism. Local autonomy and control would be retained, yet support globally optimal airlift scheduling. The optimization approach is inspired by price-directive decomposition methods that date back to the 1960s, but involve recent advances in artificial intelligence for distributed problem solving, and new wide-area communication technology.

The report is organized into three major Sections. Section 1 provides technical descriptions of the research that has been accomplished, including the client/server distributed computing model, the extended set partitioning optimization model, and the ways in which the system can be used to advantage. Section 2 is a user manual for the Dakota software system. Schedulers who wish to actually use the system in practice to schedule aircraft will make extensive use of the material in this section. There are descriptions of the various software environments and how to use them, and also a description of the underlying database system. Section 3 provides an abstract description of the database management system that was developed for the project. This material would be of basic interest to anyone who wishes to significantly expand or modify the system.

SECTION 1

AIRLIFT SCHEDULING AND DISTRIBUTED COMPUTING

I. OVERVIEW

The application of interest is the customized short-term scheduling of aircraft to support mission-critical travel. Under this project, the mission scheduling problem has been researched, and produced the associated DAKOTA software system. DAKOTA will be installed for use by U. S. Air Force Europe (USAFE) in 1994. The main components of the system are an extensive and proprietary database management system, an advanced graphical user interface (GUI), and a mathematical optimization scheduling procedure.

USAFE manages a fleet of aircraft that supports passenger travel in Europe, all of Africa, and much of the former Soviet Union. Passengers include officers and government officials carrying out missions of critical importance to the military. The DAKOTA system makes it possible to coordinate the scheduling and utilization of airlift assets in a global fashion among multiple schedulers working with a heterogeneous aircraft fleet. It is expected that utilization of the airlift resources can be significantly improved, through the operation of the new distributed optimization methods developed and integrated into DAKOTA.

1.1 Problem Description

A request for travel is specified by its point of origin, desired departure time, destination, desired arrival time, number of passengers, and mission priority. The available fleet is composed of differing aircraft types (in the USAFE problem alone, there are C-12, C-20, C-21, T-43, UH-1N and C-130 aircraft). The aircraft vary widely in several aspects, including capacity, endurance and speed. Multiple schedulers create missions that support the travel. This involves assigning contingents to aircraft, preparing flight itineraries, coordinating activities with personnel responsible for crew scheduling, and communicating with passengers being scheduled. Prominent problem characteristics are given below.

1. Several schedulers work semi-autonomously, but are allocating commonly held airlift resources.
2. Aircraft can simultaneously service multiple travel requests.
3. Travel requests have varying priorities.
4. Both immediate and advance and immediate travel requests occur.
5. Some travel requests can pre-empt others, requiring that rollback be supported.
6. Aircraft are constrained in capacity.
7. Aircraft types have differing characteristics, including airspeed, endurance, and cost.
8. There are multiple aircraft of each type.

A comprehensive system was developed to provide complete decision support for individual schedulers who work cooperatively. *Figure 1.1* illustrates this process. The requests for travel are made to the system. The large supporting database of relatively static information concerning airport runway lengths, hours of operation, aircraft capacities and operating characteristics (e.g., speed, onboard facilities), passenger histories, etc. is also accessible for the purpose of scheduling missions by the system. The Scheduler draws upon

a number of sources of information and analysis in devising a mission schedule. The Scheduler's Toolkit provides the following:

1. Several views of the existing schedules (e.g., maps, tables, *greaseboard*), to help the scheduler determine where new travel requests might fit.
2. Feasibility checking routines, that determine whether prospective bookings would get to their destinations on time, and measures of how much *inconvenience* would be incurred.
3. Schedule optimization tools, that allow the scheduler to specify sets of travel requests and (possibly empty) missions, then produce optimal and/or near optimal scheduling options.

The output is the actual missions that are scheduled.

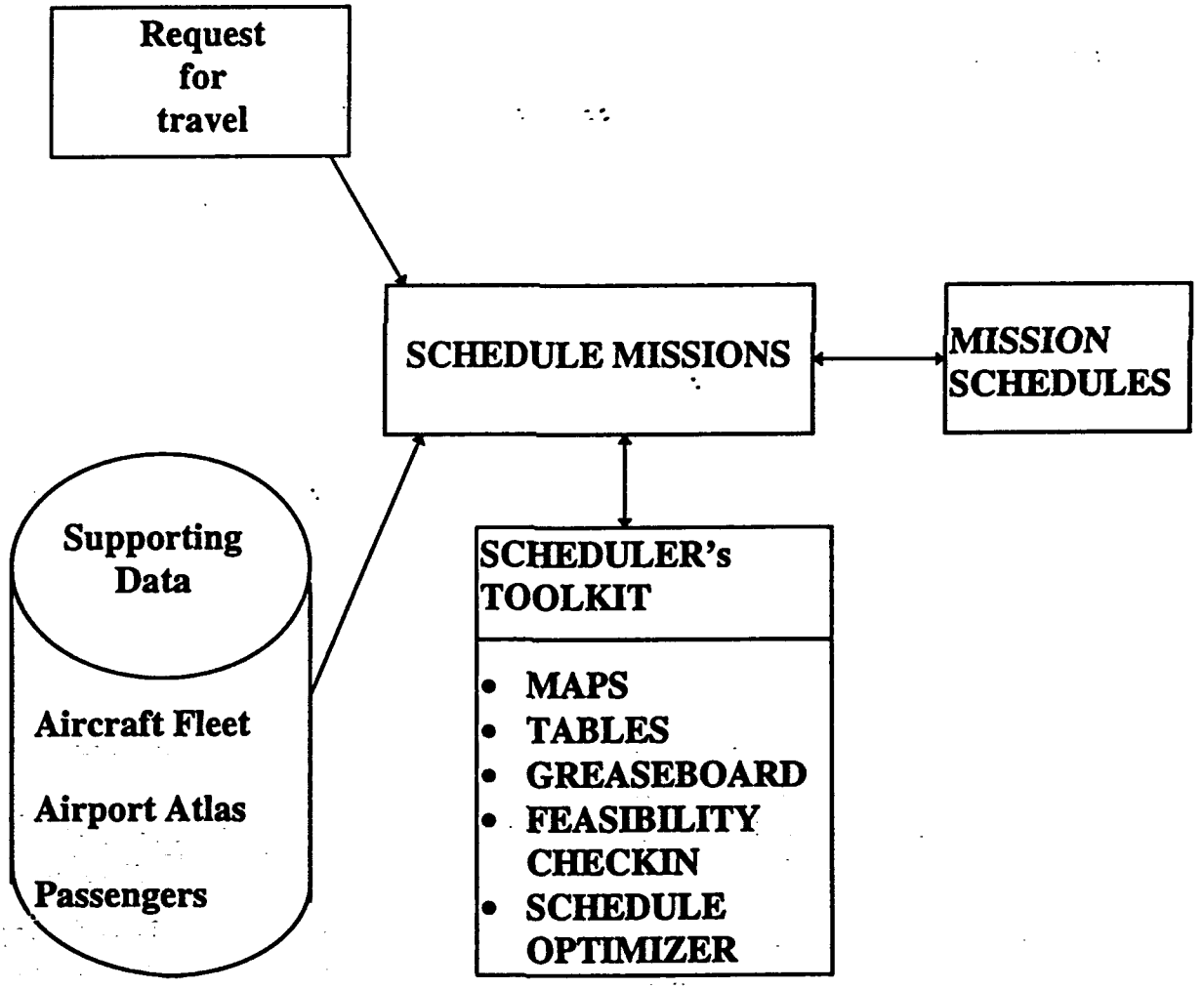

Figure 1.1 : A comprehensive view of the system to provide complete decision support for individual schedulers.

The decision support environment is inherently distributed, because multiple schedulers work in parallel, each responsible for scheduling travel requests on dedicated aircraft. From an optimization view there are several objectives: maximize the number of travel requests supported by priority class, minimize mileage traveled, and minimize inconvenience to the passengers (extra stops and waiting for example). If the schedulers work independently, they may each produce locally optimal results within their limited domains, but the coalesced work is likely to be far from optimal. In the DAKOTA system, the methodology shown in the flow diagram below is employed in supporting the scheduling process.

Step 1 Select Travel Requests and Missions for Scheduling

Step 2 Select a Request Ordering Criterion and an Insertion Location Evaluation criterion

Step 3 Order the Selected Requests Firstly by Priority Class, and, Secondly, by the Request Ordering Criterion

Step 4 Loop through the Requests in Order.
For each request and for each Mission, use Constraint Propagation to Calculate and Generate the Schedule Associated with Inserting the Request in the best Feasible Location as Measured by the Insertion Location Evaluation Criterion

Step 5 Formulate a Set Partitioning Problem from the Generated Schedules. Apply the Pricing Heuristic to Select a Subset of Schedules and Present them to the scheduler

Figure 1.2 : Flow Diagram for the Scheduling Procedure

A client-server distributed computing model was adopted to provide the means of coordinating the work of the schedulers. Within this paradigm, the computer used by each scheduler is handled as a ***client.*** The ***server*** handles access to the database, and provides mechanisms for coordinating the activities of the schedulers, with the goal of producing schedules that are close global optimality. The clients benefit, in their local decision making, from the server's global view of aircraft allocation and contention for available aircraft. The server functions range from as fundamental as ensuring that multiple schedulers do not schedule the same aircraft for conflicting missions in the same time period, to coordinating a distributed ***bidding*** process for aircraft time under contention.

Each scheduler is supported by a sophisticated mathematical optimization model that employs constraint propagation to generate the ramifications of time-windows on travel requests and aircraft availability, and a set partitioning solver [Nygard, 1993; Sycara et al, 1991; Sadeh, 1993]. These decentralized solutions are evaluated for resource contention by

the server. In cases of conflict, the server iteratively employs a pricing mechanism to encourage local alternatives that break the conflicts and ultimately induce a globally optimal solution, a process inspired by Dantzig-Wolfe decomposition [Dantzig and Wolfe, 1960].

Steps 1-4 provide a collection of candidate solutions that can be formulated in a set partitioning problem. The effectiveness of the procedure is, in part, due to the use of request ordering in Step 3, a technique that is effective in directly generating schedules in job-shop scheduling. We now present the technical basis for the components of the system.

II. RESOURCE ALLOCATION AND SCHEDULING PROBLEMS

Resource allocation and scheduling problems involve tasks which must be allocated to facilities (resources), and scheduled for processing on those facilities. The resources and the tasks to schedule may be subject to constraints that restrict allocation possibilities and the time frames during which tasks can be scheduled. This work concerns resource allocation and scheduling problems in which multiple schedulers work in parallel and schedule resources held in common. The level and structure of the coordination of the work of the schedulers is a central issue. At one extreme, the schedulers work in complete independence from each other. In this situation, the scheduling agents will produce schedules that are at best optimal only within their domain of local responsibility, and the aggregate solution may be far from global optimality. The other extreme is full communication, coordination and cooperation among all scheduler activities, the equivalent of centralized single agent scheduling. This situation permits the attainment of a globally optimal scheduling solution, but may be impractical due to the workload level being inherently distributed or too large to centralize.

Although much research and development has been done in scheduling and routing, there is little that concerns developing schedules in a distributed multi-agent environment [20]. In the case of centralized or single agent scheduling, a host of issues involving contention for resources never appear. The fundamental complications in the multiple agent case are: i) currency of information retrieved for local schedule optimization and; ii) balance of resource allocation between local and global concerns. The challenge of distributing scheduling tasks among multiple agents who share resources lies in the limited view of the current resource needs and the intentions of peer schedulers. If an agent does not have a global view of the system, a good local resource allocation decision may have a negative global impact. It may even be the case that partially completed schedules may make a globally feasible resource scheduling assignment impossible. When this occurs, requests must be unscheduled, a process called schedule rollback. Excessive rollback creates scheduler inefficiency. To address the myopic view of scheduling in the distributed case, we present a model which provides, to the individual schedulers, information about availability and system wide contention for resources. Using this global information, a human scheduler or scheduling algorithm can make local scheduling decisions which are also good globally.

In scheduling executive airlift, several schedulers are responsible for arranging travel for executives who submit requests composed of one or more origin-destination pairs, called request legs. Each scheduler is responsible for a subset of travel requests and develops aircraft routes and schedules for them subject to constraints on the aircraft fleet and on the requests themselves. Some constraints are operational in nature and are due to physical and temporal restrictions on the aircraft and crews. Travel speed, aircraft capacity, endurance, and crew duty hours are examples of these. Constraints imposed by the requests include contingent size, ***hard*** time windows on departure and / or arrival times, and ***soft*** constraints concerning inconvenience to the travelers. Hard time windows are inviolate. Soft time

windows can be violated at a penalty cost. Other constraints are imposed by the schedules developed by peers. The schedulers, working semi-autonomously, attempt to obtain the best possible schedules for the requests for which they are responsible, without violating existing constraints. Globally, the problem is to share the aircraft, possibly simultaneously, among requests for travel in a manner which optimally supports the requests.

We present an approach to single agent scheduling which characterizes the allocation and scheduling of aircraft among requests for travel as a set partitioning problem by constructing feasible assignments of travel requests to aircraft and representing them as set partitioning columns. Each column represents the feasible assignment of an ordered subset of the set of unscheduled request legs to a specific aircraft mission. Algorithmic tools for constraint propagation [7,12], variable ordering and value ordering [17] are employed to assist in manual schedule construction or to provide automatically generated solutions at the local scheduler level. Next, we focus on distributed computing models for including global information in local decision making. In this context, where the resources of interest are aircraft missions and the activities are requests for travel, we discuss three distributed computing models for resource allocation and scheduling. ***Autonomous peer process***, ***Cooperative peer process***, and ***Client-server*** models are presented in increasing order of extensiveness of the global information and coordination available to the local schedulers. For each model, mechanisms for maintaining global resource information and distributing it to the appropriate agents is described. Finally, we describe a paradigm for the distributed scheduling of aircraft missions to service travel requests which uses global information in making local scheduling assignments.

This work makes significant contributions in two areas. A constraint satisfaction based heuristic insertion algorithm is developed for single agent airlift resource allocation and scheduling. This algorithm adapts concepts from Dial-a-Ride (DARP), Job Shop Scheduling (JSSP) and Critical Path Methodology (CPM) to the problem. Methods for constraint propagation and constraint satisfaction, which are key elements in our strategy, are combined with request insertion to incrementally develop feasible schedule assignment representations. These representations may be evaluated by a set partitioning heuristic to produce schedules which meet operational objectives while satisfying constraints on both the aircraft and on the requests. The algorithm is extended to a multiple scheduling agent paradigm supported by a distributed model which facilitates system wide propagation of global information and provides a mechanism for effectively communicating resource requirements and contention. Through this model, we contribute to an improved understanding of the means of maintaining, communicating and utilizing global resource information to optimize schedule construction in a distributed setting. This decision support environment is representative of varied problem solving applications that can benefit from cooperative allocation and scheduling of resources, such as job shop scheduling and other manufacturing problems.

Software development and testing for this work is being done on DEC 5000 and Sun SPARC 10 Unix workstation computers. The software is portable to any Unix system adhering to the proposed Open Systems Foundation (OSF) standard. Code is written in

ANSI C and C++. Database support for the maintenance of requests, scheduled missions, and supporting information is provided by DESc [9], a locally developed database management system for scheduling applications. The inter-process communication uses flow-controlled, connection-oriented Unix network sockets. The network software has been extended to manage a larger software system for scheduling which includes a graphical user interface employing the X Windows network protocol. The platform for the application is a network of Unix color graphics workstations employing the TCP/IP network protocol suite. Work on this project is being done at North Dakota State University under Air Force Office of Scientific Research (AFOSR) sponsorship through a subcontract with Netrologic Incorporated.

III. DEFINING ARASP

The definition of the Airlift Resource Allocation and Scheduling Problem (ARASP) is motivated by a need to identify the salient features of the executive airlift scheduling process. Number of scheduling agents, level of agent cooperation and coordination, request (task) characteristics and constraints, and resource characteristics and constraints are feature categories which characterize the structure of the problem.

3.1 Scheduling Agents

For small enterprises it may be reasonable for a single agent to schedule all travel requests. However, larger enterprises require that multiple scheduling agents share the task; each scheduler assuming responsibility for scheduling a subset of the requests for travel. Agents may be located in close proximity to each other, so that personal interaction can guide the sharing of resources. On the other hand, agents and resources may be geographically separated within an installation or even throughout the world. In order to be truly flexible in our problem solution, we assume that multiple schedulers, linked by a computer network, work in parallel to service distinct requests by allocating commonly held aircraft resources.

3.2 Support for Cooperative Scheduling

The efficiency of individual schedulers and the global effectiveness of the schedules they create are primary concerns in the design of computer support for the allocation and scheduling process. The individual schedulers and the scheduling group, as a whole, benefit from the sharing and coordination of information regarding system wide resource availability and contention. Two identifiable elements of computer support for the system are:

- A computer network model should be implemented to encourage sharing of global information regarding resource availability, coordinate global resource allocation and support cooperative resource use.
- Algorithmic support at the level of the individual scheduler should provide optimization and decision support tools for schedule construction and for evaluation of performance measures of resulting schedules.

3.3 Request Characteristics

Travel requests may be composed of multiple request legs (departure / arrival pairs), but generally represent a single contingent. Each request leg has associated time window information, described in terms of earliest/latest pickup and earliest/latest delivery. The bounds on these windows may be designated as soft or hard, depending on if they are negotiable or inviolate, res.

The contingent size may vary as individual passengers join or leave the group between stops on the itinerary, but when physically possible, the entire contingent will travel on the same aircraft. When contingent size precludes service by one craft, it is assumed that multiple requests will be submitted. For passenger convenience and security, it is considered infeasible to require a contingent to change aircraft during a request leg. Individual request legs, however, may be supported by different aircraft. Multiple requests (contingents) may be simultaneously supported by the same aircraft. Thus, stops during a request leg are permitted.

Requests are both immediate and advance notice. They may have varying priorities, which can influence the order in which they are scheduled. Since operational goals may allow pre-emption and since cancellations may occur, a means for unscheduling previously supported requests and rollback of schedules must be provided.

3.4 Resource Characteristics

The fleet of supporting aircraft may be non-homogeneous, composed of aircraft with varied capacity, air speed, and endurance. For the specific application, we consider small aircraft in the six to fifteen seat category. These operational constraints and others regarding flight and service time must be enforced, though there should be means to override them to provide "what-if" capability in decision support. Operational regulations regarding the length of time an aircraft may remain in service, required maintenance schedule, and flight, duty and rest time for crews suggest constraints on aircraft that can best be reflected by specifying our resource to be an aircraft mission. The aircraft mission becomes a loosely defined mechanism for ensuring that an aircraft returns to its home base at required intervals and allows for controlling the amount of time it is in service, in maintenance, specially allocated or unavailable for general scheduling. By focusing on the aircraft mission as our resource, we also provide improved granularity for viewing and manipulating the fleet schedule.

The above characteristics and constraints describe the paradigm for which we propose heuristic algorithms that may be used as decision support tools or as tools for automated scheduling. We then propose a model for cooperative resource allocation and scheduling in which global resource information is maintained and distributed to individual scheduling agents. In addition to providing resource information, the model provides a means for enabling negotiation for resources and for insuring the integrity of global resource schedules.

IV. AN ENVIRONMENT FOR COOPERATIVE SCHEDULING

Globally, the goal of our system is to efficiently allocate and schedule resources to optimally support a set of travel requests in a manner which is consistent with the constraints defined by the enterprise and by the customers. To achieve this goal, our system maintains a representation of the evolving constraints on available resources and provides this information to agents in a distributed scheduling environment. The two key elements in the system are algorithmic support for the individual scheduler and coordination of a scheduling effort distributed among multiple schedulers. Fundamental to our approach to both of these concerns is the characterization of reso availability in terms of the constraints placed on those resources by the enterprise, by t sks for which support has already been scheduled and by resource requirements of individual schedulers for meeting the demands of the tasks they must schedule. We form a constraint hierarchy whose root represents constraints on the entire aircraft fleet and whose leaves are constraints on individual travel request legs.

At the global level, we maintain information, in the form of fleet, aircraft and mission constraints. These constraints represent the flexibility of resources to accommodate additional requests for travel or temporal schedule shifts. They are a function of operational considerations and of constraints on request legs already being supported by the given mission. Temporal constraints can be intuitively represented as time windows on the activities of the resource for supporting the tasks assigned to it. This global information is provided, system wide, to schedulers as they consider resources to meet the needs of their unscheduled travel requests.

Individual schedulers select resources, guided by aircraft characteristics, customer needs and preferences, resource availability and schedule flexibility. The schedulers are required to request a resource and an associated feasible time window before making schedule changes to it. This provides a mechanism for determining the availability of a resource and for informing peer schedulers of resource use intentions. It also imposes additional constraints in the constraint hierarchy which will protect the reserved mission and associated reserved time window while the scheduler is making changes to it at the local level. By imposing these additional constraints, the local scheduler is essentially asking for a lock on a resource over a commonly agreed upon period of time. Only after the reservation is granted, can the local scheduler be certain that changes to the local schedule of the resource will not affect nor be affected by changes to the rest of the hierarchy. It is this step that bridges the gap between the individual, autonomous scheduler and the group of cooperative schedulers who share resources. *Figure 4.1* illustrates the propagation of constraints throughout the constraint hierarchy, both within and between levels.

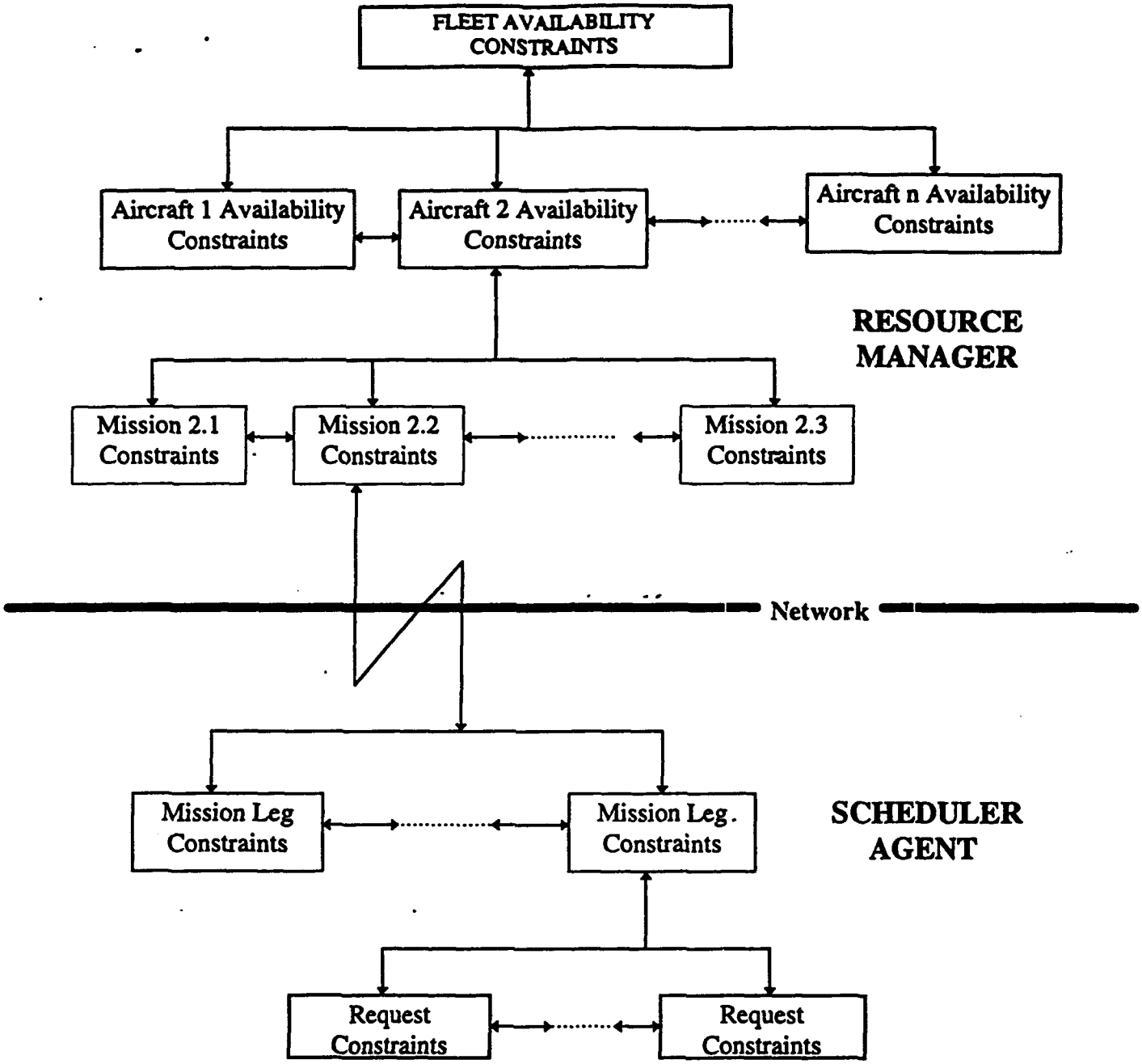

Figure 4.1: Propagation of time window constraints in ARASP. Effects of constraints are propagated up and down the hierarchy, as well as laterally through a level.

Resources made available at the individual scheduler level and the requests the agent must schedule onto them are also characterized by their constraints. The resources reserved by the scheduler cannot be extended beyond the constraints negotiated at the time they are acquired. Feasible assignments of tasks to resources involves producing a schedule which simultaneously satisfies the constraints on both the task and the resource. Thus, the search for feasible, and ultimately optimal, local schedule assignments is directed by those constraints imposed on the resources and on the tasks. The degree to which the distributed

model imposes restrictions on the acquisition of resources and their associated time windows determines the involvement of global decision making in the local scheduling process. Potential exists to provide a spectrum of control from the simple locking approach described above to an autocratic system that dictates which resources should be utilized to service subsets of travel requests. Our approach relies on the cooperative nature of the schedulers to negotiate reasonable time windows for their reserved resources.

The propagation of constraints in the constraint hierarchy is pursued further in the discussion of support for the individual scheduler. There we give an overview of the algorithmic support for local determination of resource schedules.

V. SUPPORT FOR THE INDIVIDUAL SCHEDULER

Whether there is a single scheduling agent or multiple cooperative schedulers, we view the actual assignment of a task to a resource as being done by an individual agent. To simplify our discussion, we concentrate on the temporal restrictions due to mission time windows and request leg time windows. The foundation of our scheduling process in either mode, therefore is a mechanism for a single agent to schedule a selected set of request legs onto a set of time constrained aircraft missions.

Schedules for resources may be incrementally constructed over time, with actual scheduling done to meet specific lead time requirements of some tasks. The philosophy that guides our approach to scheduling of request legs in the executive airlift problem is that previously scheduled requests should be minimally impacted by the addition of new, unscheduled, requests to a common mission. With this as a primary constraint on creating request / mission associations, our heuristic uses an insertion approach to create assignments of request legs to missions which do not violate time window conditions on existing request leg originations and terminations. Barring violation of capacity constraints, a request leg may be serviced by an existing mission leg, or it may be necessary to create additional stops in the mission schedule to accommodate the request. Time window bounds for arrivals and departures of individual request legs may be categorized as ***soft*** or ***hard***. A hard time window bound is one whose violation will cause an infeasibility in the constructed mission. Insertion of mission legs to service request legs which would violate existing time window constraints are deemed infeasible and are not considered further by the algorithm. Violation of soft time window bounds produces inconvenience to the customers, but does not cause infeasibility. A request leg assignment causing a mission to violate a soft window bound for one or more of its request legs will incur a penalty for doing so, which will reduce the offending assignment's value.

Given a set of resources (aircraft missions and corresponding operational constraints) and a set of tasks (unscheduled flight requests composed of one or more request legs), the single agent scheduling algorithm performs the following activities:

- Orders request legs for insertion, based on schedule performance considerations,
- Generates Set Partitioning Problem (SPP) columns representing feasible ordered insertions of requests into aircraft schedules and their associated values, and
- Solves the Set Partitioning Problem to produce updated aircraft schedules.

The ordering of the travel request legs may be viewed as preprocessing, while the Set Partitioning Problem solution performs a postprocess function. The heart of our algorithm focuses on the propagation and satisfaction of constraints to generate representations of feasible task-to-resource assignments. Here, we focus on the ordering and column generation components of the process.

5.1 Variable Ordering

Travel requests can be scheduled as they are received or may be short-term or long-term batched. Since insertion algorithms are inherently myopic, the order in which insertions are performed is critical to the quality of the resulting schedules. Batching requests allows the application of variable ordering [17] techniques to prescribe a request leg order for insertion consideration. The particular ordering employed depends on the operational and strategic goals of the scheduling effort. ***Most Constrained First*** ordering insures that those requests which will be difficult to schedule get early consideration, when the schedules for the resources are still most flexible. ***Best Fit*** ordering is an attempt to increase utilization of resources on flight legs which are already scheduled. Those requests which may be serviced by existing flight legs are considered for insertion early, with a preference given to those which will most nearly fill an aircraft for the request leg path. ***Earliest First*** ordering can be used to delay scheduling of those requests farther out on the scheduling horizon and thereby maintain flexibility of schedules until more requests for that time frame are in hand. These schemes may be combined or used separately to address those measures of performance deemed most critical to the enterprise.

The order given by the variable ordering process to the set of unscheduled request legs under consideration is the one which will dictate the order in which scheduling attempts for the requests will be made by the insertion heuristic. Our insertion algorithm views this ordered set of request legs as a sequence and determines subsequences of it which may be feasibly assigned to each resource.

Once a request leg order has been determined for scheduling, the insertion process can begin. If it is determined that a particular request leg may be supported by a given mission, the point of insertion into the existing schedule remains to be determined.

5.2 Column Generation

At the local scheduler level, mission schedules are represented as event lists. Each event corresponds to a take-off or landing of the aircraft, either for service or to support a travel request. To each event there corresponds time window constraints (ET, LT). These bounds can be viewed as an extension of the constraint hierarchy, propagated down from the time constraints on the mission and up from the time windows on the requests being serviced by the event. The ET and LT values for an event are created and maintained in a manner similar to the forward pass and backward pass for maintaining ET and LT values in the Critical Path Method (CPM) [11]. Each insertion of a travel request onto a mission results in possibly creating new events, adding the constraints imposed by the new request to those already in place, and recalculating the ET and LT for events in the event list, using the forward and backward method of CPM to propagate constraints across the bottom level of the constraint hierarchy. A time window becomes violated by an insertion when it produces an ET value for an event which is greater than its corresponding LT value. Only those insertions which do not violate any of the existing time windows for events are

feasible. If an insertion of a request is deemed feasible, value ordering is performed on the values associated with all feasible insertions of the request leg into the resource schedule.

The generation of feasible task-to-resource assignments can be viewed as a constraint satisfaction problem. Assignments which are feasible are exactly those which can be performed without violating the constraints on the task, on previously scheduled tasks, or on the resource. From the sequence of unscheduled request legs, a SPP column and associated cost are generated for each subsequence which may be feasibly assigned to a specific resource. As assignments of tasks to resources are considered, the effects of time windows for the new tasks propagate to those of existing flight legs. This alteration of flexibility of aircraft flight legs affects the time window for the entire aircraft mission, within the allowable bounds of the resource. The effects of committing to a particular change in a mission schedule will propagate to the time windows of other missions for the same aircraft and ultimately to the schedule for the entire fleet. Likewise,alterations in the scheduling constraints at the fleet level affect the time windows for take-offs and landings to accommodate a particular request leg. Thus the fleet schedule structure may be viewed as a dynamic model, in which alterations at any level propagate throughout the entire system, affecting constraints on each component. Only those alterations that preserve the satisfaction of all previously established constraints are considered feasible. Figure 1 illustrates how the insertion of a request leg on a mission can ultimately affect the entire fleet schedule..

The upper levels of the hierarchy in *figure 4.1* represent information which must ultimately be available globally in a distributed paradigm. We assume, in our insertion heuristic, that the schedule and allowable time window for a specific mission are a local concern at the point when scheduling takes place. This allows the scheduling agent the autonomy needed to manipulate the schedule locally within constraints imposed at the time the mission is selected for scheduling. It further allows us to develop an insertion heuristic for local scheduling which may be extended implemented within a distributed scheduling system.

VI. DISTRIBUTED COMPUTING MODELS FOR ARAS

A model for decentralized airlift resource allocation and scheduling (DARAS) must provide basic information support needed for the individual schedulers to perform their tasks. Moreover, integrity of data must be maintained as reservations are created and edited by multiple schedulers. The model must provide a means of performing distributed scheduling which is both efficient for the agent and effective in producing globally desirable schedule structures. Three basic models that provide varied facilities for information sharing are discussed. These are referred to as the "Autonomous peer process", "Cooperative peer process", and "Client-server" models. These three models are listed in order of increasing suitability for implementing a distributed decision support system for multi-agent scheduling. We discuss them in general and then make recommendations for DARAS, in particular.

6.1 Autonomous peer processes

In this model all schedulers execute identical processes which allow them to enter new requests for travel, create/edit reservations for aircraft and generate reports by directly accessing the underlying database, as if each scheduler was the exclusive scheduling agent. Acquisition of available aircraft resources is permitted on a first come, first served basis, and individual schedulers have no information provided by the software support environment concerning resource needs of other schedulers. The advantage to this model is that algorithmic extensions from the single agent case are simple. This model is sometimes employed when multiple schedulers are stationed in close proximity to each other, and use verbal agreements to prevent much contention from arising. A disadvantage is a need for a sophisticated database management system to maintain data integrity during concurrent use of the system by multiple schedulers. Heavy contention for resources will cause inefficient scheduling and may cause scheduling roll-backs, due to lacking or incomplete knowledge of concurrent scheduling efforts. The lack of a global view limits possibilities for providing meaningful algorithmic decision support in the scheduling process.

In order for individual scheduling processes to be informed of potential contention for a given aircraft, communication between all processes must be established and information about resources of mutual interest must be shared between the appropriate processes. Lacking this communication, efforts to minimize schedule rollback will require a locking mechanism to reserve aircraft, thereby precluding other agents from accessing them. Since the scheduling process may take considerable time, locking is not scalable and the model was rejected as an alternative for the current work. Requiring that each scheduling agent communicate scheduling intentions with every other scheduler places an undue burden on both the schedulers and the communication network. It is clear that providing mechanisms for sharing of resource requirement information between schedulers requires a model in which relevant data is easier to collect, maintain, and disseminate.

6.2 Cooperative peer processes

This model was proposed by Sycara, et. al. [20] for the distributed job shop scheduling problem. In that environment, shop "stations" or machines, each able to service a single requested task at a time, are scheduled to complete "orders" composed of requirements for processing by one or more stations. The basis of the model is the establishment of a mechanism which allows processes to communicate resource needs and scheduling intentions with other processes that share a need for a particular resource.

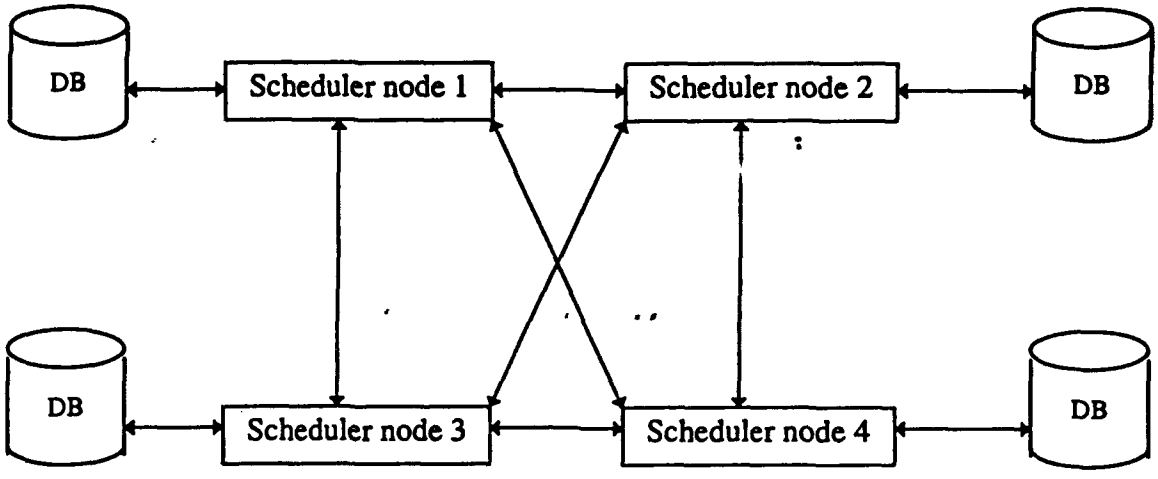

Figure 6.2.1: An example of the connections required for a fully connected four-node cooperative peer process network

In terms of airlift scheduling, where a resource may service several requests simultaneously, we make each scheduler process a "node" in a distributed scheduling system. Each scheduler agent becomes the resource manager of a set of aircraft and is the only agent enabled to make changes to the schedules of those aircraft it controls. An agent considering employing a particular aircraft, in a schedule being constructed locally, must request current information on the availability of that resource from its manager, and in so doing must "register" an interest in that aircraft. The resource manager then updates the aggregate demand information for that particular aircraft and provides it to all registered scheduling agents. Finally, any scheduler requiring a resource must send a request to the resource manager who, upon receipt of a reservation request, verifies that the aircraft is available, makes appropriate database accesses to update schedules, and then shares the updated information with the "interested" schedulers. This model can clearly be implemented on a single multi-processing machine, but can also be extended to a network of computers.

As an example, *Figure 6.2.1* shows the communication connections for a fully connected network of four scheduler nodes. The number of connections maintained is O(n2), where n is the number of participating nodes. Communication can be minimized by only communicating information from a resource manager to those nodes which have indicated

an interest in a particular aircraft. The model also makes fewer demands on the database management system, for concurrency control, by requiring that only an agent responsible for a resource can update its schedule and related information. The underlying database can be either centralized or distributed. Support for local decision making with attention to global resource needs becomes achievable. However the implementation of global optimization strategies are difficult because there is no view which encompasses all aircraft schedules. The loss of communication with one or more of the nodes in the network makes the resources maintained by the node(s) inaccessible to the rest of the schedulers. This is not likely in the single machine, multiple process case, but becomes an issue in the decentralized case. The problem can be mitigated by having backup nodes which take over the "down" nodes responsibilities, which further complicates the model.

6.3 Client-server model

In this model, the server is a process that waits to be contacted by a client process, and then services the client's request. In the context, of the executive airlift application, the client processes are the scheduler application processes, and the server acts as a resource manager. The server maintains allocation and contention information about aircraft, seats and times each craft is available, providing it to those scheduling agents requesting a particular resource. It controls access, maintains aggregate demand information, and communicates with "interested" nodes for all aircraft.

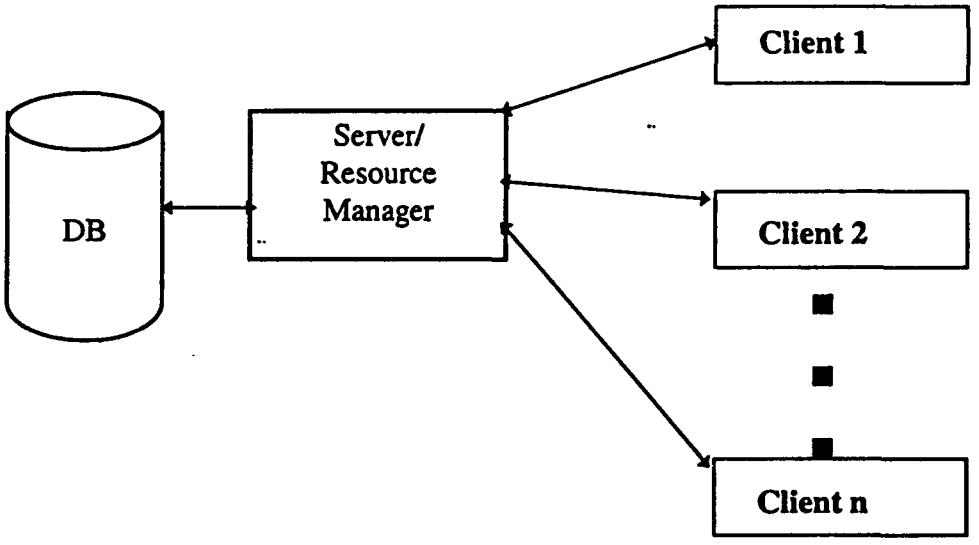

Figure 6.3.1: The client server model with n (scheduler) clients.

As with the cooperative peer processes model, scheduling agents are required to register intentions with the server before being granted access to or information about a particular aircraft and its schedule. Reservation requests for updating an aircraft schedule are sent to the resource manager, which has sole access to the database and can provide update information to those schedulers concerned with that particular resource. As *figure 6.3.1* indicates, the number of communication connections in this topology is O(n) for a model

with n clients. The server process can provide either iterative or concurrent service to their clients. Each type has advantages and disadvantages, depending on the application.

6.4 The Iterative Server

The iterative server queues messages from clients and processes them in a FIFO manner. This model works well when it is known that processing a message takes a small, relatively constant amount of time. If the number of simultaneous accesses to the server increases with the number of clients, this model will not scale well. Long waits in queue or refused messages will degrade performance at client nodes. However, there are advantages to this model, if the client access load is relatively low, as is the case when clients obtain resource information from the server and then do optimization locally. The implicit serialization of messages from clients through a single server access to the data base requires a more simple form of concurrency control in the DBMS. The iterative server handles all requests for resources and distributes all information on resource availability and contention and is therefore in an ideal position to maintain a current view of resource allocation. This type of server is well suited for the implementation of global optimization strategies in distributed systems.

6.5 The Concurrent Server

A concurrent server, upon receipt of a client request, begins a new (child) process, *dedicated exclusively to the service of the request.* Upon completion of its tasks, the child process terminates. The parent server process is therefore available to accept requests from clients while other requests are being serviced. In this way it is possible to service multiple client requests. In the case of a large number of client processes attempting simultaneous access to the server, this model will, on the average, begin the service to a client in a more timely fashion. Database accesses will, however, occur from one of multiple server clones and will require greater concurrency control than the iterative server model. Furthermore, unless a mechanism for sharing up-to-date information between the server clones is implemented, management of a global view of resource availability becomes more difficult, and hence it is difficult to provide global optimization support. This can be mitigated by the implementation of shared memory access and a global optimizer process accessed by all the server clones.

6.6 The Client Server Paradigm for DRAS

Of the three models investigated, the client-server model provides for the most information being distributed to appropriate nodes with the least amount of communication overhead. In the airlift scheduling environment, where we expect a relatively small number of agents working simultaneously, the iterative server is the most appropriate, due to the natural way in which global optimization strategies can be implemented. Inherently a network model, the client-server model can be naturally implemented on either a single machine or a network of machines. The communication protocol for this model on a single machine is simple because communications failures between processes are not an issue. When extending to a network, the protocol must take into account failure at a node and network failure, though node failure is not as damaging as in the cooperative peer process model. If the server is expected to make all database accesses, the protocol must be designed to accommodate a wide range of client requests and an equally wide range of server responses. These can include complete aircraft reservation schedule structures, for all aircraft, for a given schedule period. A significant challenge to implementing this model is providing the server and clients with strong decision support and optimization strategies.

6.7 Implementation of the Client Server Model for DRAS

A fundamental motivation for choosing the client server model is the desire to provide a global view to the scheduling process and to control access to commonly utilized resources. Database accesses by clients, either for information about resources under consideration or for database update, are therefore a natural point at which to introduce the separation of client and server responsibilities. The client process continues to function at the local level as it does in the single agent paradigm, however all database activities are now directed to the server process. By empowering the server with ultimate control of database accesses, we insure both database integrity and the ability of the server to maintain a global view of the demand for resources.

In designing the single agent model, we anticipated the extension to a networked environment and consequently, we implemented a design which would seamlessly incorporate a network interface. The clients' database access function calls are intercepted by a network interface which redirects them to the server process. Figures 6.7.1 a) and b) demonstrate the insertion of the network interface and server process into the direct database access for the autonomous scheduler.

Once the network interface has been established, it is possible for the single scheduling agent client to make the same database requests and updates that were made directly. Now, however, the server is responsible for intercepting those requests and making accesses to the database. This puts it in a position to coordinate allocation of shared resources and propagate updated information regarding resource availability and constraints to its clients. The level of coordination provided by the server may vary from simple advisory information on resource availability and contention to autocratic determination of specific resource assignments. At minimum, a reservation system, or locking mechanism, must be

provided for the server to insure that multiple schedulers are not concurrently scheduling the same resource. To facilitate resource locking, a lock table is provided to the server and additional message types for client-server communication are added.

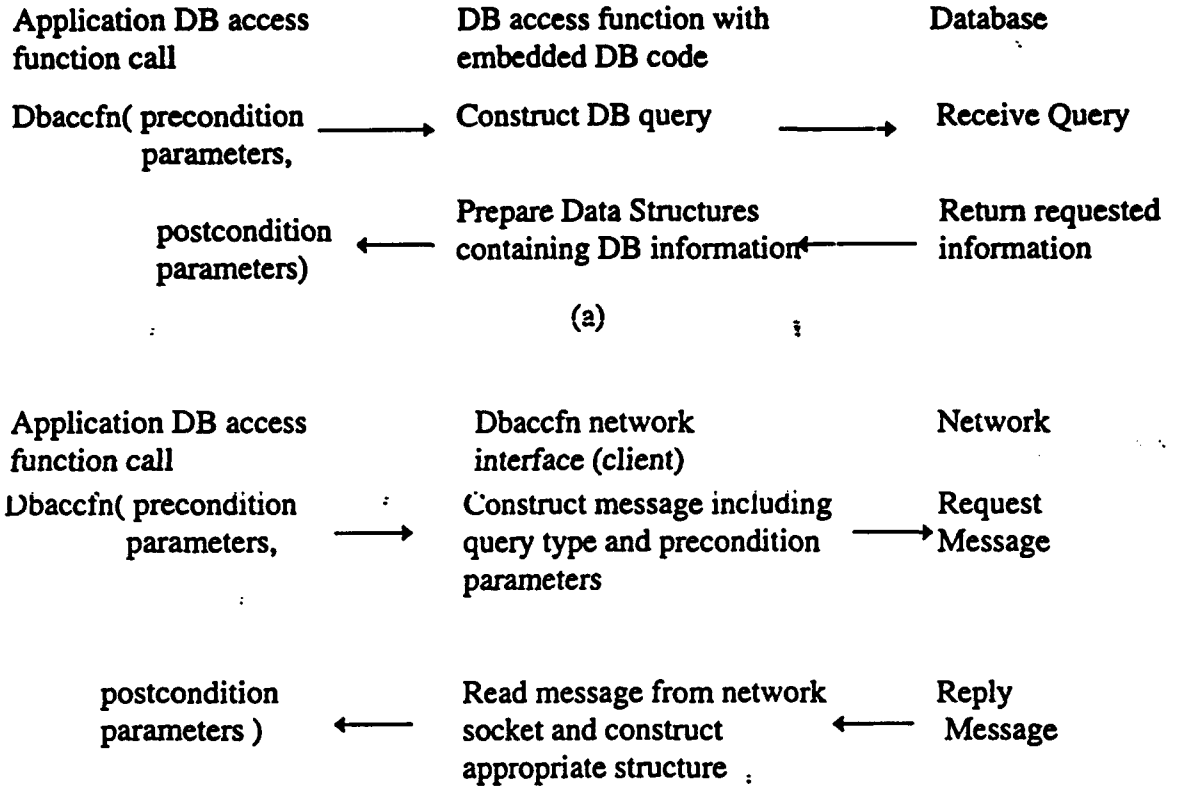

Figure 6.7.1: The unobtrusive extension of the local single scheduler application to the distributed paradigm involves inserting client and server network interfaces between the application's DB access function call and the DB access function.

As indicated above, the roles of the server may vary, depending on the desired balance between local autonomy and central control in the scheduling effort. As an advisor, the server can provide its clients with information which should help them make good local decisions based on knowledge of peer needs, and depend on scheduler altruism to guide equitable allocation of resources among individual agents. In its role as negotiator, the server can negotiate directly with a client over resource requests based on the server view of global resource needs or it can act as an intermediary when resource contention arises.

6.8 Propagation of Constraints in DARAS

The propagation of constraints in the distributed environment is as previously described for the case of cooperative schedulers. The server in the client-server model, having assumed the role of resource manager, now maintains the representation of the schedules for the fleet, aircraft and individual missions. When a scheduling agent requests a particular mission for scheduling purposes, that agent must negotiate with the server for a time window within which the mission under consideration must remain. This time window is constrained by bounds imposed by the enterprise, by time windows allotted to other missions or reserved by other schedulers for the same aircraft, and by the time windows for previously scheduled requests within the mission. Likewise, the time window granted to the scheduling agent may further constrain the expansion of other missions for that craft. The scheduling agent then applies local scheduling heuristics to obtain a mission schedule. After the local scheduling task has been completed, the effects of the scheduling process are propagated back to the server, which updates the global schedule representation and makes the updated representation available to other scheduling agents.

VII. DISTRIBUTED SCHEDULE OPTIMIZATION

By assigning management responsibility for a resource to a particular node in the network, we provide for efficient sharing of information about the availability of that resource, and also make it possible to provide decision support at the global level. The resource manager can be as simple as a lock manager, insuring that a scheduling agent has access to specific resources. In addition, it can function in the role of arbiter, providing guidance and information in a process of successive refinement of resource allocation. The resource manager can administer a process in which client nodes iteratively determine what resources are worth to their local schedules and ***bid*** on them.

In the airlift scheduling problem, the individual schedulers can solve local scheduling problems and then, when contention for an aircraft is detected by the resource manager, determine the incremental cost of an alternative solution that does not require that resource. Each local scheduling heuristic can register a value associated with accepting an alternative solution. The resource manager, at this point, can assign new ***purchase*** values to the resources and scheduling begins anew, with adjusted resource values. The Dantzig-Wolfe decomposition process is an example of an approach similar to this from the linear programming literature [5].

Our current implementation of the distributed scheduling process is between the two extremes. We use the resource manager to maintain control over the use of resources so that multiple schedulers do not simultaneously have access to a particular time slice for an aircraft. The resource manager negotiates the time slice with a scheduler, viewing the demands of the other schedulers. The scheduler then works within the constraints imposed by the resource manager to schedule a set of request legs onto a set of aircraft missions.

The local scheduling heuristics for schedule maintenance, column generation and set partitioning, as described earlier are applied to the sets of missions and unscheduled request legs obtained from the resource manager. Prior to algorithm assisted scheduling, the scheduler may manually make insertions. The tools for maintenance of mission schedules and detection of infeasibility work equally well with manual scheduling as with algorithm assisted scheduling and provide the scheduler with valuable information regarding feasibility and performance of scheduling decisions.

The scheduling heuristics provide a human scheduler with alternatives for scheduling. The ultimate power to manually schedule request legs, within the constraints imposed by the resource manager, and/or to accept all or part of the heuristic solution lies with the human scheduler. The same insertion functions used in the insertion heuristic are also available to a scheduler to provide assistance in directly creating a schedule with editing tools.

Once the local schedule is created, a request to alter the missions involved is sent by the scheduler to the resource manager. It is the responsibility of the resource manager to make update accesses to the data base and to propagate the effects of the altered missions to the

other missions of the affected aircraft, between the aircraft in the fleet, and ultimately to the fleet schedule.

The schedule optimization algorithm is designed to provide the best service to unscheduled requests with minimal disruption for requests already scheduled. An insertion algorithm propagates the effects of supporting a new travel request on existing pickup and delivery time windows within a set of missions. A companion algorithm is used to construct the set of feasible assignments of requests to aircraft and pickup/delivery times. To help ensure that difficult-to-schedule legs are given ample opportunity, a prioritization scheme is employed to attempt insertion of legs in decreasing order of scheduling difficulty. The result of the application of the insertion heuristic on a set of request legs and a set of aircraft missions is a set partitioning matrix representing all feasible assignments of request legs to aircraft. A solution to the set partitioning problem is generated via a pricing technique inspired by the simplex method for linear programming, and applied to the matrix to provide scheduling suggestions to the human scheduler. These solutions may be accepted, by the scheduler, in whole, in part or discarded. This mechanism allows the human scheduler to inspect the effects of various scheduling strategies and choose the one best suited to the situation. A set partitioning solver based on a pricing procedure is used to obtain the solution to the optimization model.

In a set partitioning model, the goal is to identify a collection of variables (columns, representing aircraft missions) that partition the rows (representing travel requests) so that the objective function value is minimized. Since the number of possible columns that could be generated is exponential in the number of travel requests, it is important to develop criteria that restrict the number of columns generated, yet retain columns that represent good schedules.

A Basic Set partitioning formulation is given below:

$$\begin{aligned} \max \quad & px \\ \text{subject to :} \quad & Ax = 1 \\ & 0 \le x \le 1 \\ & x \text{ all integer} \end{aligned}$$

where

x : the decision vector of bits. A one indicates that the corresponding route is traveled or that the corresponding delivery is not made

p : the profit row vector.

A : a zero/one matrix with the following form:

1 : a vector of all 1s

	Route Columns	Non Delivery Columns
Aircraft	**B**	**0**
Deliveries	**C**	**I**

Submatrix **B** consists of columns of 1s and 0s, with each column corresponding to a candidate solution in which routes are assigned to aircraft. Assignments correspond to 1s. Non assignments correspond to 0s. In submatrix **C**, a 1 in a column means that a delivery is made, the 0 means the delivery is not made. **0** is a matrix of all zeroes. **I** is the identity matrix. **A** has one row for each aircraft and one row for each potential delivery. **A** has one column for each route and one column for each potential delivery. The equalities involving the aircraft rows mandates that each aircraft is used precisely once. The equalities involving the delivery rows mandates that each delivery will be made by precisely one aircraft or not delivered at all.

Following the principle that a aircraft can feasibly do less, if A has a column for a aircraft making some set of deliveries, then it also has a column for that same aircraft making each subset of those deliveries.

An extended set partitioning problem (ESSP) is stated mathematically as follows :

Indices

i = 1,2,.....,n aircraft
j ε J(i), the set of feasible schedules for aircraft i.
k = 1, 2, ...,l requests for travel

Data

c_j = cost of schedule j

v_{kj} = 1 if schedule j supports request k, 0 if not

p_i = penalty cost for an idle aircraft

q_i = penalty cost for overscheduling an aircraft

r_k = penalty cost for not accommodating a travel request

s_k = penalty cost for overaccommodating a travel request

Decision Variables

$y_j = 1$, if schedule j is selected, 0 if not

$u_i = 1$, if aircraft i is not utilized, 0 if utilized

$\sigma_i \geq 0$ and integer, indicating overscheduling of aircraft i

$w_k = 1$, if request k is not supported, 0 if supported

$\alpha_k \geq 0$ and integer, indicating multiple support for request k

Mathematical Formulation

$$\text{Min} \sum_j c_j y_j \sum_i (p_i u_i + q_i \sigma_i) \sum_k (r_k w_k + s_k \alpha_k)$$

Subject to:

$$\sum_{j \in J(i)} y_j + u_i - \sigma_i = 1 \qquad \text{for each aircraft i}$$

$$\sum v_{kj} y_j + w_k - \alpha_k = 1 \qquad \text{for each travel request k}$$

$$y_j, u_i, w_k = 0 \text{ or } 1$$

$$\sigma_i, \alpha_k \geq 0 \text{ and integer}$$

A pseudo code of the algorithm for finding a set of best routes is given below.

```
findBestRoute(aircft, allRoutes, routesPicked) {
/* tries to find a more profitable route for aircraft aircft if successful it updates routesPicked
   to reflect the improvement for this to work right the profit a non-delivery must be zero.
  aircft is the aircraft to be rerouted
  allRoutes is a representation of the A matrix
  routesPicked is an array of routes traveled by aircrafts
  curRoute is the tentative replacement for aircft
  incProfit is the amount of improvement
  rP2 is the tentative replacement for routesPicked
  delSet2 is the updated deliveries made by another aircraft after some have been usurped
          by aircraft aircft
  newR is the corresponding route
*/

  route = routesPicked[aircft]

  for each curRoute such that aircft travels route curRoute {
    incProfit = value( curRoute ) - value( route )
    rP2 = routesPicked;

    for each aircraft aircft2 except aircft {
    if ( incProfit ≤ 0 ) break ;
      delSet2 = {all deliveries made by route rP2[aircfts]} -
            {all deliveries made by route curRoute }
      newR = route traveled by aircraft aircft2 making deliveries
          delSet2
      incProfit += value( newR ) - value( rP2[aircft2] )
      rP2[aircft2] = newR
    } /* aircfts */

    if ( incProfit > 0 ) {
      routesPicked = rP2
      routesPicked[aircft] = route = curRoute
    }
  } /* curRoute */
} /* findBestRoute */
```

Solutions are generated very quickly, allowing many alternatives to be considered by the schedules in real time. The overall process of generating aircraft schedules using all the tools developed is shown in *figure 7.1.*

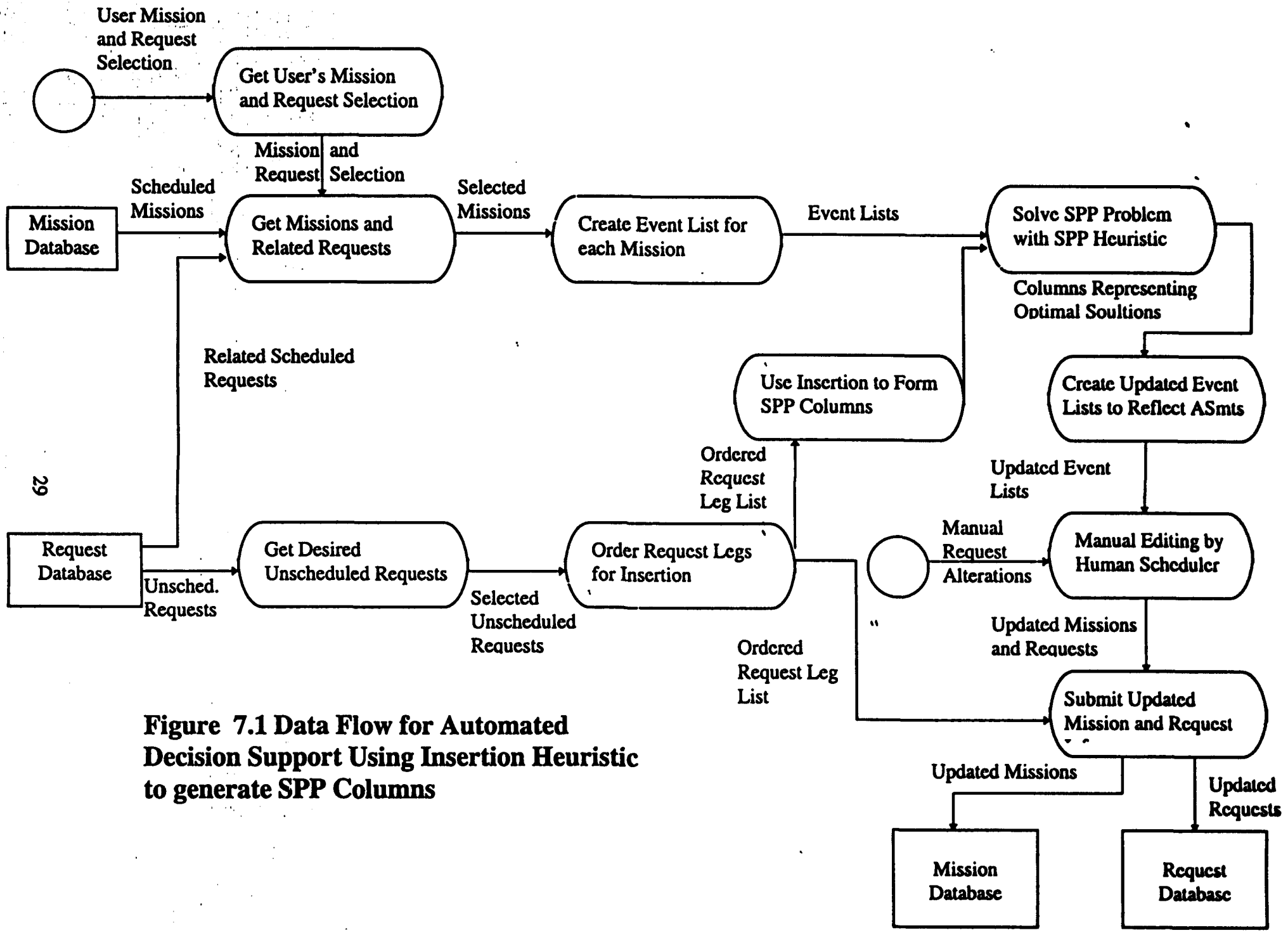

Figure 7.1 Data Flow for Automated Decision Support Using Insertion Heuristic to generate SPP Columns

VIII. CONCLUSION

We have described a means of representing resource schedules as a hierarchical structure of constraints. This representation provides a individual scheduler with a representation of the resource allocation and scheduling problem for which consequences of task insertions are intuitive to maintain and propagate. Further, we have shown how this model provides a basis for communication of resource availability and contention, through the use of constraint propagation, in an environment composed of geographically separated cooperative scheduling agents.

We have presented three network models providing varied levels of support for scheduler cooperation and have defended our choice of the client server model as a paradigm for distributed resource allocation and scheduling. Our current implementation of the client server model provides us with a paradigm for increasing the server involvement in the scheduling process to include negotiations between scheduling agents contending for a commonly desired resource. An extended set partitioning model for generating actual schedules was also presented.

The single agent scheduling algorithms have been thoroughly tested and provide stable performance and good quality solutions for problems involving up to 20 aircraft missions and up to 50 travel requests. We are continuing our work at the distributed level to provide our server with the greater capabilities we have presented in this work.

IX. REFERENCES

1. Appelgren, L. H. A Column Generation Algorithm for a Ship Scheduling Problem. Transportation Sci., 3:53-68, February, 1969.

2. Appelgren, L. H. Integer Programming Methods for a Vessel Scheduling Problem. Transportation Sci., 5:64-78, February, 1971.

3. Bell, Colin E. and Austin Tate. Using Temporal Constraints to Restrict Search in a Planner. Technical Report, Artificial Intelligence Applications Institute, University of Edinburgh, December, 1984.

4. Brown, Gerald G., Glenn W. Graves and David Ronen. Scheduling Ocean Transportation of Crude Oil. Management Science, Vol. 33, No. 3, March, 1987.

5. Dantzig, G. and P. Wolfe, "Decomposition Principle for Linear Programs," Operations Research 8(1), 101-111, 1960.

6. Darby-Dowman, K. and G. Mitra, An extension of Set Partitioning with Applications to Scheduling Problems. European Journal of Operations Research 21, 1985

7. Fox, Mark S.,"Constraint-Directed Search": A Case Study of Job Shop Scheduling, PhD dissertation, Computer Science Department, Carnegie-Mellon University, 1983.

8. Fox, Mark S. and Stephen F. Smith. ISIS: a Knowledge-Based System for Factory Scheduling. Expert Systems 1(1):25-49, July, 1984.

9. Ganapathy, Prabukumar, DESc: A Database Engine for Scheduling Applications, MS Thesis, Department of Computer Science and Operations Research, NDSU, Fargo, ND, 1993.

10. Garfinkel, R. S. and G. L. Nemhauser. The Set Partitioning Problem: Set Covering with Equality Constraints.Oper. Res., 17:848-856, 1969.

11. Hillier, Frederick S. and Lieberman, Gerald J., Introduction to Operations Research, Holden-Day, Inc., 4432 Telegraph Avenue, Oakland, CA 94609, 1986.

12. LePape, C. and S. F. Smith, "Management of Temporal constraints for Factory Scheduling", Proceedings IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems (TAIS 87), Elsevier Science Publishers, held in Sophia Antipolis, France, May 1987.

13. Marsten, R. E. An Algorithm for Large Set Partitioning Problems. Management Sci., 20:774-787, 1974.

14. Miller, David, James Firby and Thomas Dean.Deadlines, Travel Time, and Robot Problem Solving. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence. Los Angeles, California, August, 1985.

15. Rit, Jean-Francois.Propagating Temporal Constraints for Scheduling.In Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, Pennsylvania, August, 1986.

16. Ronen, D.Cargo Ships Routing and Scheduling: Survey of Models and Problems.European J. Oper. Res., 12:119-126, 1983.

17. Sadeh, N. and Fox, M. S., "Focus of Attention in an Activity-based Scheduler," Proceedings of the NASA Conference on Space Telerobotics, 1989.

18. Smith, Steven F. and Peng Si Ow,"The use of Multiple Problem Decompositions in TimeConstrained Planning Tasks", in Proceedings of the Ninth International conference on Artificial Intelligence, 1013-1015, 1985

19. Smith, Stephen F., Peng Si Ow, Claude Le Pape, Bruce McLaren and Nicola Muscettola. Integrating Multiple Scheduling Perspectives to Generate Detailed Production Plans.In Proceedings of the SME Conference on AI in Manufacturing. Long Beach, California, September, 1986.

20. Sycara, K., S. Roth, N. Sadeh, M. Fox, "An Investigation into Distributed Constraint-directed Factory Scheduling", Proceedings - The Sixth Conference on Artificial Intelligence Applications, Santa Barbara, California, 1990.

SECTION 2

USER MANUAL FOR

THE DAKOTA SCHEDULING SYSTEM

I. INTRODUCTION TO DAKOTA

The DAKOTA software has been designed to facilitate the mission routing and scheduling tasks for the United States Air Force in Europe. The system facilitates scheduling of mission critical travel by distinguished visitors and officers, serving Europe, the former Soviet Union, and Africa as well as transcontinental flights to the United States. A heterogeneous fleet of aircraft including C-12, C-135, C-20, C-21, and T-43 planes and helicopters is available, with differing capacities, endurance, speed and facilities. Each type of aircraft is allotted a fixed number of flying hours. The objective of the process is to maximize overall efficiency of the fleet of aircraft by supporting travel of as many eligible personnel as possible, within constraints on fleet size, flying hours and other limitations.

The software allows requests for travel and information about airports, aircraft, and passengers to be entered. Missions can be scheduled, either manually or by utilizing the decision support module. A number of tools are provided to facilitate the scheduler's task, including maps, tables, feasibility checking, and summaries of available flying hours.

1.1 *MAIN WINDOW*

Figure 1.0 shows the Main Menu, which contains the following options:

- Enter Request
- Schedule
- Reports
- Input
- Utilities
- Exit

Figure 1.0 The screen for the main window appears as shown here.

The first step in utilizing the software is entry of supporting data pertaining to airports, aircraft, and passengers. This information will require only occasional updates. **Input** on the Main Menu produces another pull-down menu with options for adding data to the Airport Atlas, the Aircraft Data, and the Passenger Data. *Figure 1.1* shows the pop up menu for Input.

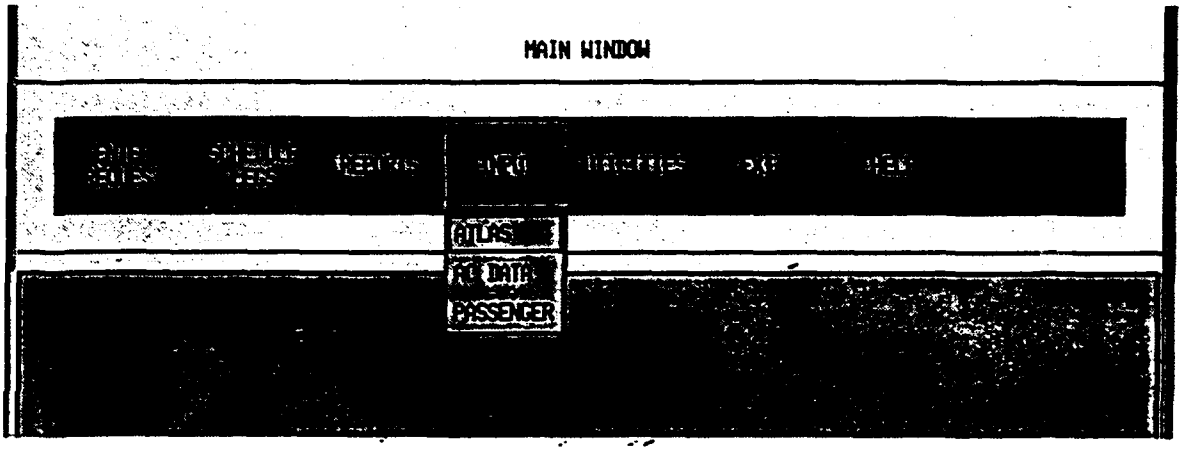

Figure 1.1. The click on the input button pulls up this pop up menu. The options in this pop up are selected by clicking the first button on the mouse on any one of them.

Next, requests for travel may be entered into the system as they arrive. The **Enter Request** option on the Main Window is selected for this purpose. This capability will allow the schedulers to forecast their needs with respect to aircraft and available flying hours, even if the request has not been formally scheduled and assigned an aircraft. At the appropriate time, requests can be officially scheduled and assigned a particular aircraft.

Scheduling can be done manually, with the itinerary and times specified by the scheduler, or by using the decision support system, which suggests appropriate aircraft, departure and arrival times. The scheduler can accept the system's suggestions or reject them and manually schedule the request.

The following sections describe each screen in more detail.

II. REQUEST SCREEN

The Request Screen is arranged in four sections as shown in *figure 2*. The first section is for entry of general information about the request. Passenger information is entered into the middle section, while information about the itinerary is entered on a leg-by-leg basis in the third section. Finally, near the bottom of the screen, general remarks and notes about the airlift request can be entered.

2.1 Adding A Request

To add a new request, the **Add** button at the bottom of the screen must be selected with the left mouse button. The general information pertaining to the request must be entered first. The agency making the request (Requesting Agency) is entered, along with the date when the request was received (Date of Req). If a particular type of plane is needed or appropriate for the task, the plane type can be entered (Req Plane Type). The point of contact (POC) is the person to call regarding the details of the request. Both the home and office phone numbers for the point of contact may be entered (Home Phone, Off Phone). The system will generate a unique number (Request ID) for this request, and the user is not allowed to change this identification number.

This screen allows the user to enter information from the Airlift Request Form, including the following:

- HEADER INFORMATION:
    - Requesting Agency
    - Date of Request
    - Requested Plane Type
    - Point of Contact
    - Home Phone for the Point of Contact
    - Office Phone for the Point of Contact
- PASSENGER INFORMATION
    - Last Name
    - First Name
    - Title
    - Social Security Number
    - Phone Number
- PROPOSED ITINERARY INFORMATION
    - Departure ICAO
    - Earliest Departure Date, Time

- Latest Departure Date, Time
- Indication of *firmness* of the Departure Time
- Arrival ICAO
- Earliest Arrival Date, Time
- Latest Arrival Date, Time
- Indication of *firmness* of the Arrival Time
- Priority Code
- Number of Passengers
- Remarks pertaining to the request

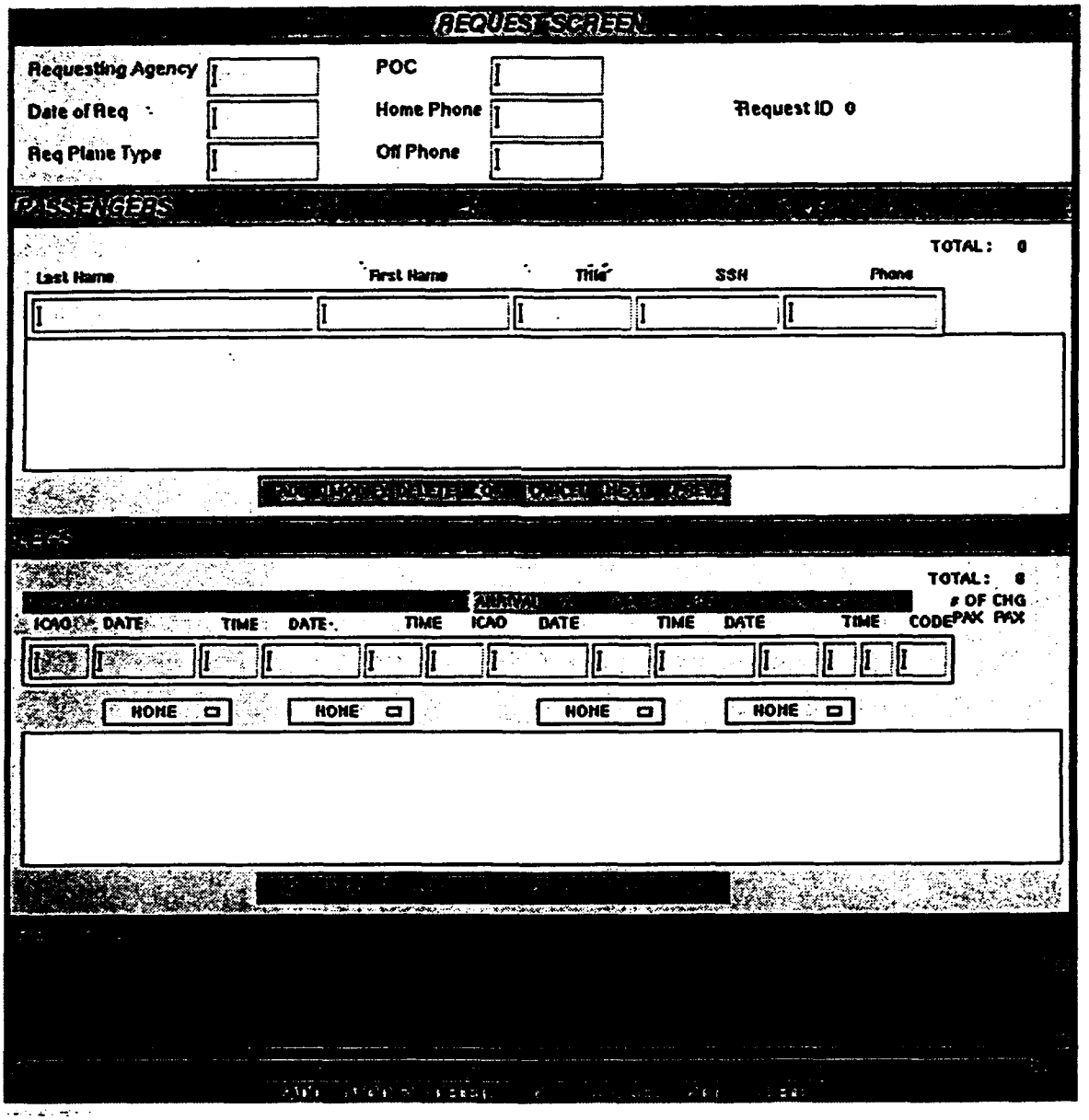

Figure 2. The request screen is divided into four sections as shown above. Initially when pulled, the buttons at the bottom are active and are brighter than the rest of the buttons.

2.2 *Passengers*

The passenger section provides a place to enter the name (Last Name, First Name), title, social security number (SSN) and phone number for passengers who appear on the request. The leftmost buttons at the bottom of the passenger section of the screen are used to indicate whether a passenger is to be added, modified, or deleted. If the passenger name cannot be entered for security reasons, a DV code can be typed into the Last Name area. When the mode is Add, the entire passenger name need not be entered. For example, if only a **T** is entered into the last name field, followed by a return, and only a return is entered into the first name field, a pop-up window appears listing all existing passengers whose last name begins with T, and if a name is selected, all of the information about that passenger is retrieved into the request screen. Alternately all the records can be selected by pressing return for both Last Name and First Name. *Figure 2.1* illustrates the pop-up window for retrieving an existing passenger's information.

After the user types information about one passenger, and presses the **OK** button, the information is moved to the area directly below the typing area. The **CANCEL** button is used to abandon the current passenger entry. If more than one screenful of passengers appears, **NEXT** and **PREV** buttons provide the capability of viewing succeeding and previous screens of passenger information. The total number of passengers entered for the request appears in the upper right corner of the passenger section.

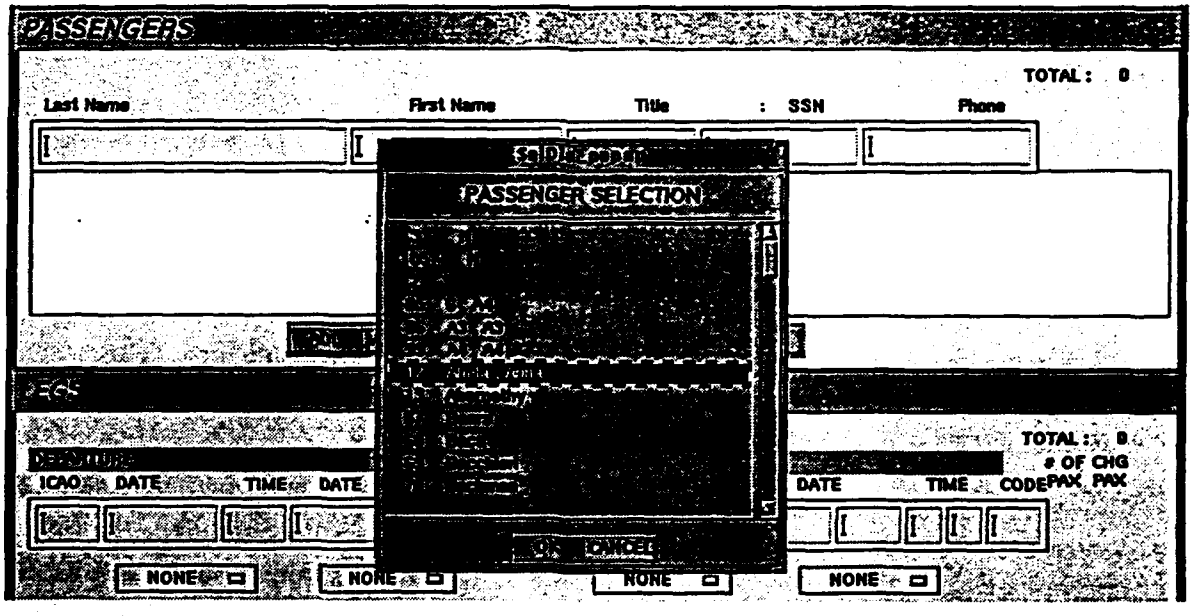

Figure 2.1. The figure shows the result of pressing a return without any entry in the Last Name of First Name field. This indicates selection of all the records. The selected name is indicated with a dashed box around the entry.

Information regarding the proposed itinerary is entered on a leg-by-leg basis. For each leg, the departure and arrival information and the number of passengers as well as the priority code are entered. If specific passengers were entered into the passenger section, the number of passengers (# OF PAX) field will reflect the number of passengers entered. If a DV code was the only entry in the passenger section, the number of passengers (# OF PAX)field must be manually entered.

2.3 Icao Selection

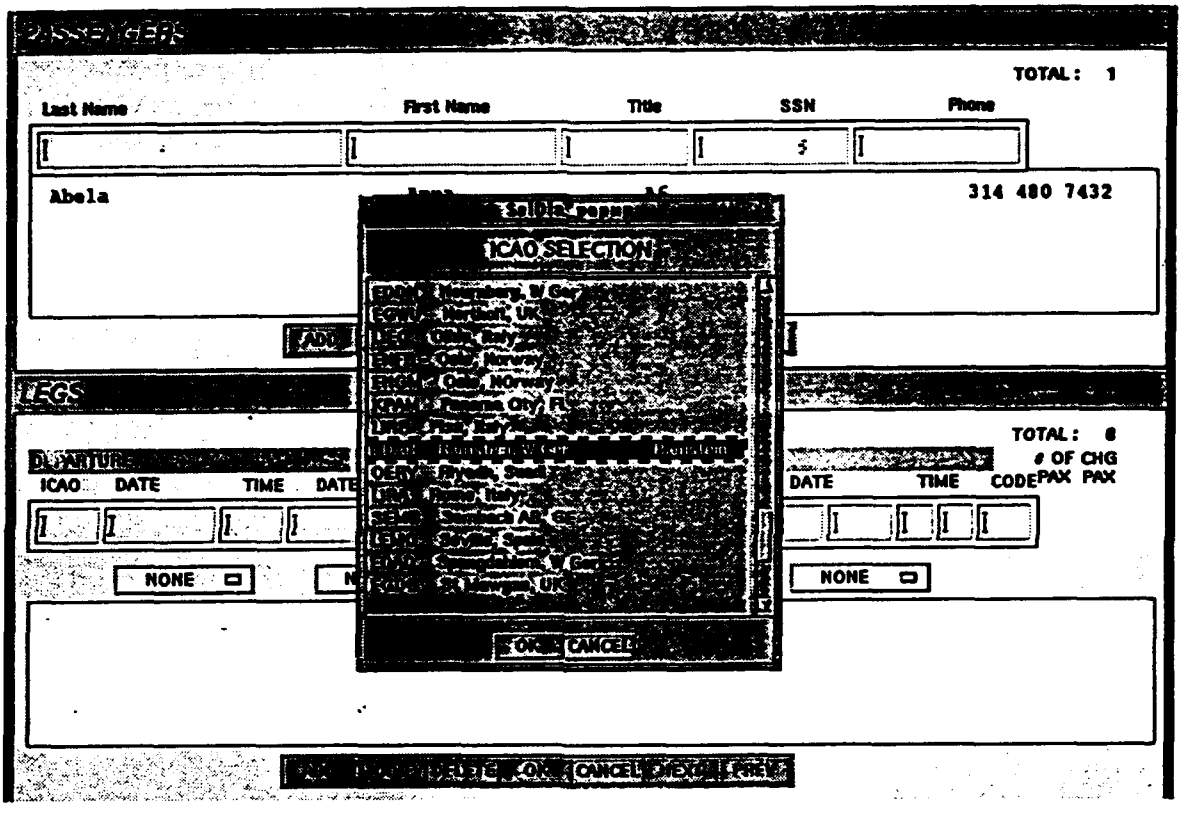

Figure 2.2. The pull down menu has all the records in the dialog box. This is done by pressing return without an entry in the ICAO field. The highlighted record is then selected by clicking on it as shown in the figure.

For each leg, the airfield identification code for the departure and arrival destinations are entered in the ICAO fields. If the ICAO is unknown, the pop-up is activated for assistance. *Figure 2.2* shows the screen after a return was simply entered into the departure ICAO. The left mouse button is used to select the appropriate ICAO. For both departures and arrivals, an earliest date/time and a latest date/time are entered in the DATE TIME fields in the form DD/MMM/YY. Under the DATE/TIME fields, there are 2 boxes labeled NONE. These boxes provide the capability of indicating the degree of flexibility in the dates and times entered. For instance, if the earliest departure time has been entered as 800, and the requesting agency has indicated that under no circumstances can the departure occur before this time, the NONE box should be changed to **HARD**. If, however, the 800 earliest departure time is flexible, the NONE box should be changed to **SOFT**. Leaving the choice

as NONE indicates the scheduler / requesting agency does not care if the earliest and latest departure times are changed.

The last section of the screen (REMARKS) is used to indicate purpose of the request and other information relevant to this particular request.

After the 4 sections of the screen have been filled in, the buttons at the very bottom of the screen allow the request to be saved or canceled. The OK button saves the request while the CANCEL button abandons the information.

2.4 Delete/Modify

The request screen can also be used to modify or delete existing requests. If the **DELETE** or **MODIFY** button is chosen from the row of buttons at the bottom of the screen, the user is prompted for either the Request ID, Lead Passenger, or Point of Contact. ***Figure 2.3*** shows the screen after MODIFY has been selected. The appropriate request will appear on the screen for modification or deletion.

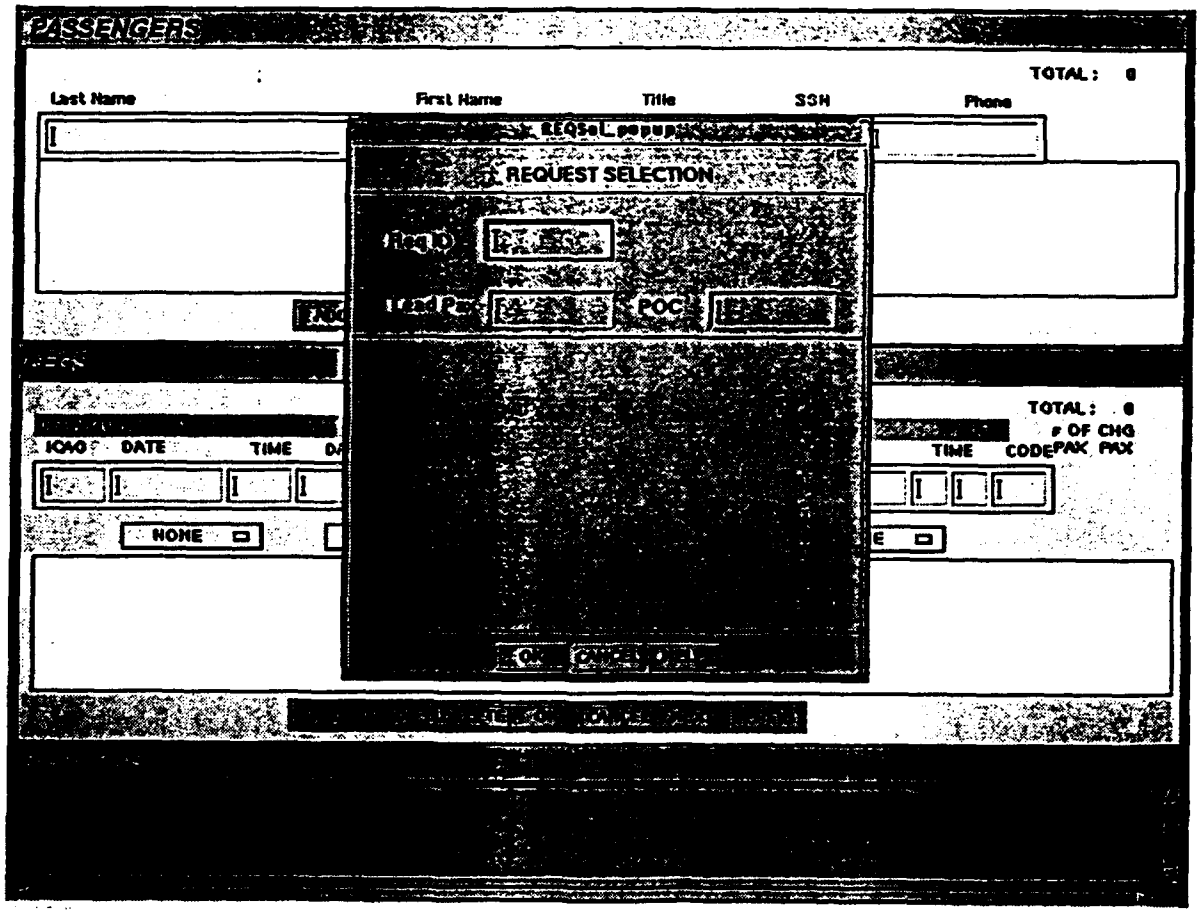

Figure 2 . The pop up when MODIFY is selected. After the entries are done in the fields labeled Req ID, Lead Pax and POC, the matching records are displayed, of which any one can be selected by clicking on it.

III. SCHEDULE REQUEST SCREEN

From the Main Menu, the **Schedule** option is selected to schedule a request to be flown by a particular plane. The Schedule Request screen is divided into four panes, as shown in *figure 3*. The upper left pane contains the map, while the panes down on the right are used for display of unscheduled requests and scheduled missions for a particular date. The lower left window either displays flying hour information or passenger information, depending on the view chosen.

After an date is entered in the upper right corner (using the *DD/MMM/YY* format), appropriate information about the requests and missions is retrieved and displayed on the screen. The map shows scheduled missions using a solid line, while unscheduled requests are shown using a dotted line. The missions are requests are color-coded, and the box in the upper right corner of the map window lists IDs and tail numbers in the appropriate colors for associating the routes on the map with the textual information along the right side of the screen.

In the upper right portion of the screen, the unscheduled requests for the date shown appear. For each request, the request ID is shown along with leg-by-leg information including date, origin and destination ICAOs and times, and the number of passengers on that leg.

The lower right area of the screen displays information about missions that have been scheduled for that date, as well as a listing of planes that are in maintenance and planes that have been reserved for that date. For scheduled planes, the Mission ID and plane type are shown along with information about each mission leg, including date, origination and destination ICAOs and times and the number of passengers on the leg.

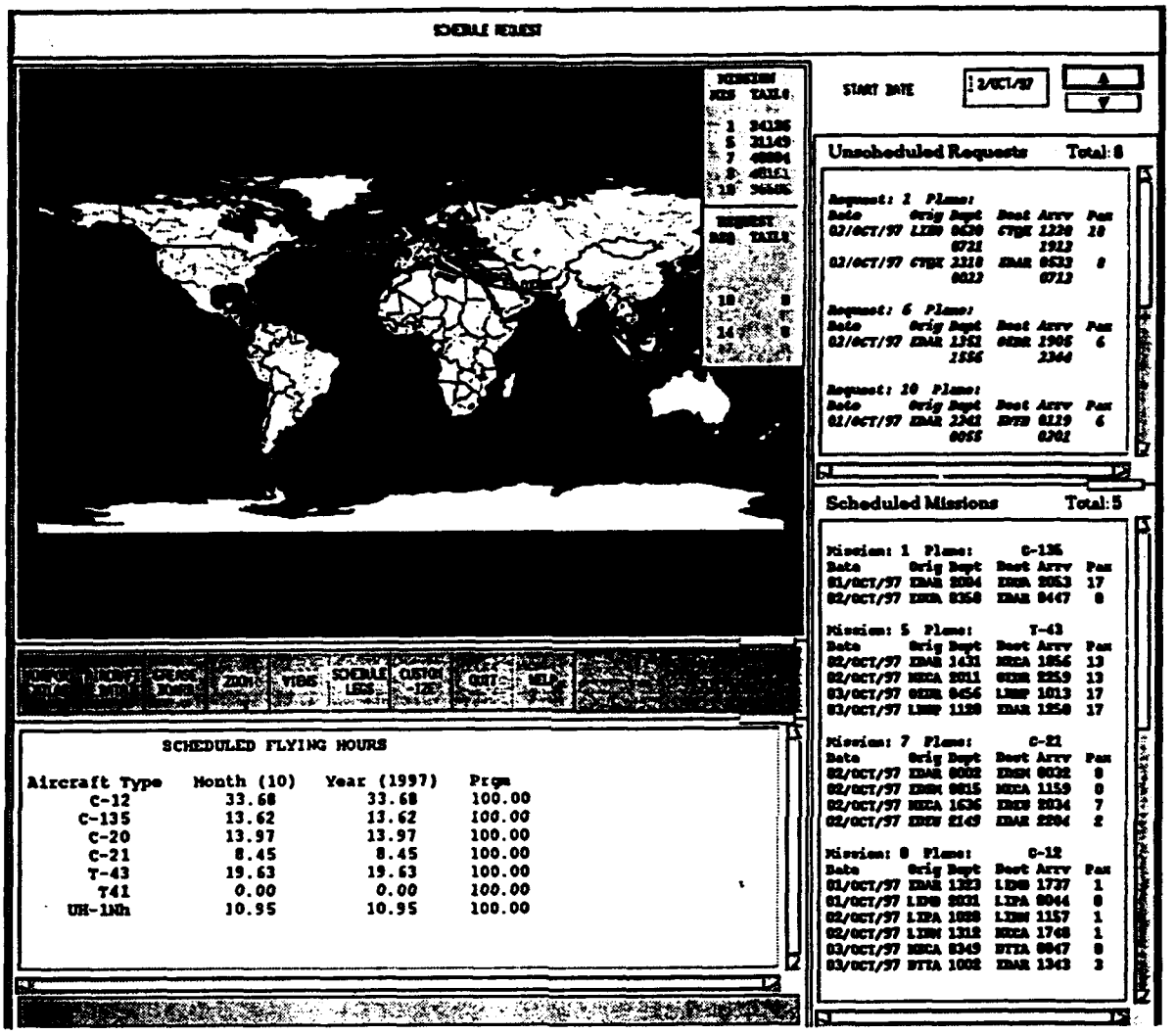

Figure 3. The Schedule screen for October 2, 1997 is shown. Note the screen is divided into four areas (Map, Flying Hours, Unscheduled Requests and Scheduled Missions). The date is entered in the DD/MMM/YY format.

3.1 Options

Buttons along the center of the screen provide various utilities for the scheduler, designed to assist in the selection of the plane, as shown in *figure 3.1.* The buttons are listed below and explained in subsequent paragraphs. The left mouse button is used to select each button, while the right mouse button is used to select from any subsequent pull-down menus.

- Airport Atlas
- Aircraft Data
- Greaseboard
- Zoom
- Views
- Schedule Legs
- Customize
- Quit
- Help

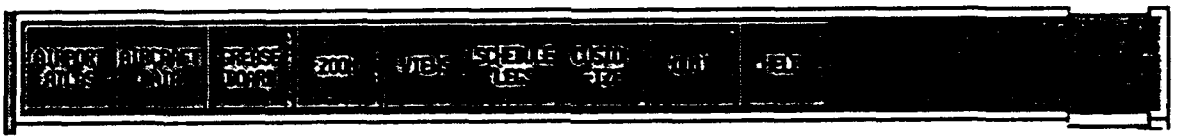

Figure 3.1. The above figure shows all the utilities for the scheduler which is helpful in the selection of the planes.

3.2 Option Airport Atlas

The **Airport Atlas** button displays a screen as shown in *Figure 3.2*, allowing for retrieval of information about particular airports.

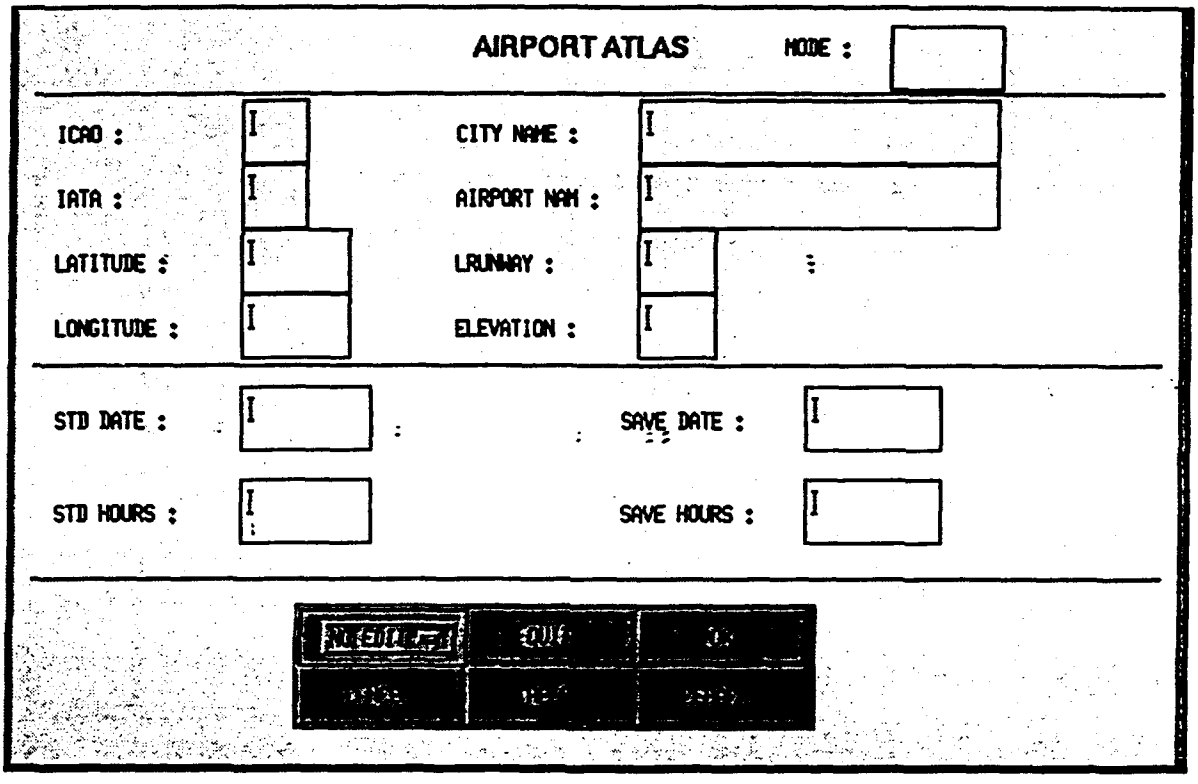

Figure 3.2. Airport Atlas Screen. This window allows retrieval of data about the various airports whose database is maintained.

3.3 Option Aircraft Data Entry

The **Aircraft Data** button displays a screen as shown in *Figure 3.3*, allowing the scheduler to display information about particular aircraft (such as speed, endurance, and capacity).

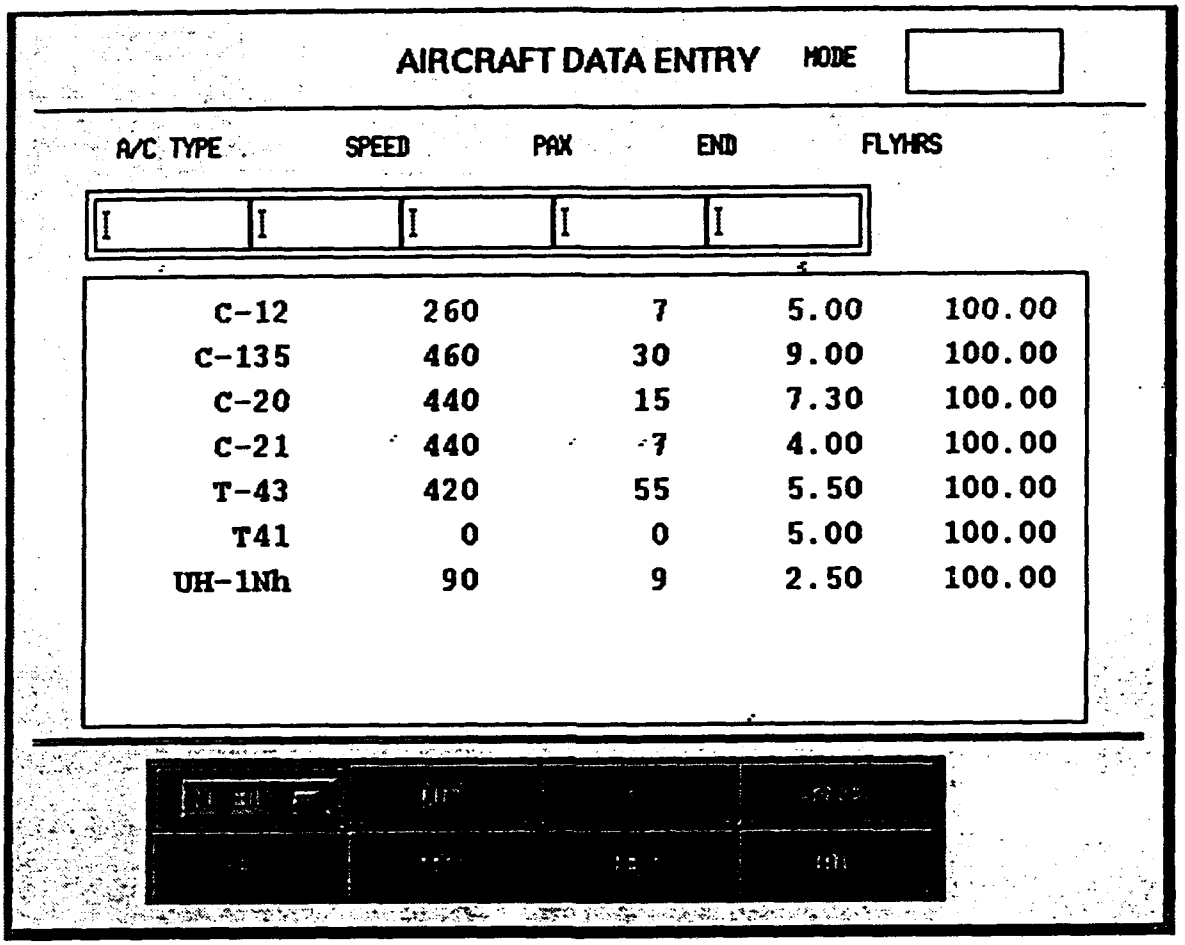

Figure 3.3. The aircraft data entry screen aids in the manipulation of data on the aircraft.

The Aircraft Data Entry Screen, Aircraft Atlas Screen and Passenger Screen are discussed in detail later in section four.

3.4 *Option Greaseboard*

The Greaseboard button displays the activity of all planes over a range of dates selected by the scheduler. Dates are shown across the top of the screen, while each plane is listed down the left side of the screen. Each entry in the grid shows the itinerary for missions (if they exist) for a particular plane on a particular date. *Figure 3.4* shows an example of the greaseboard output for a selected range of dates.

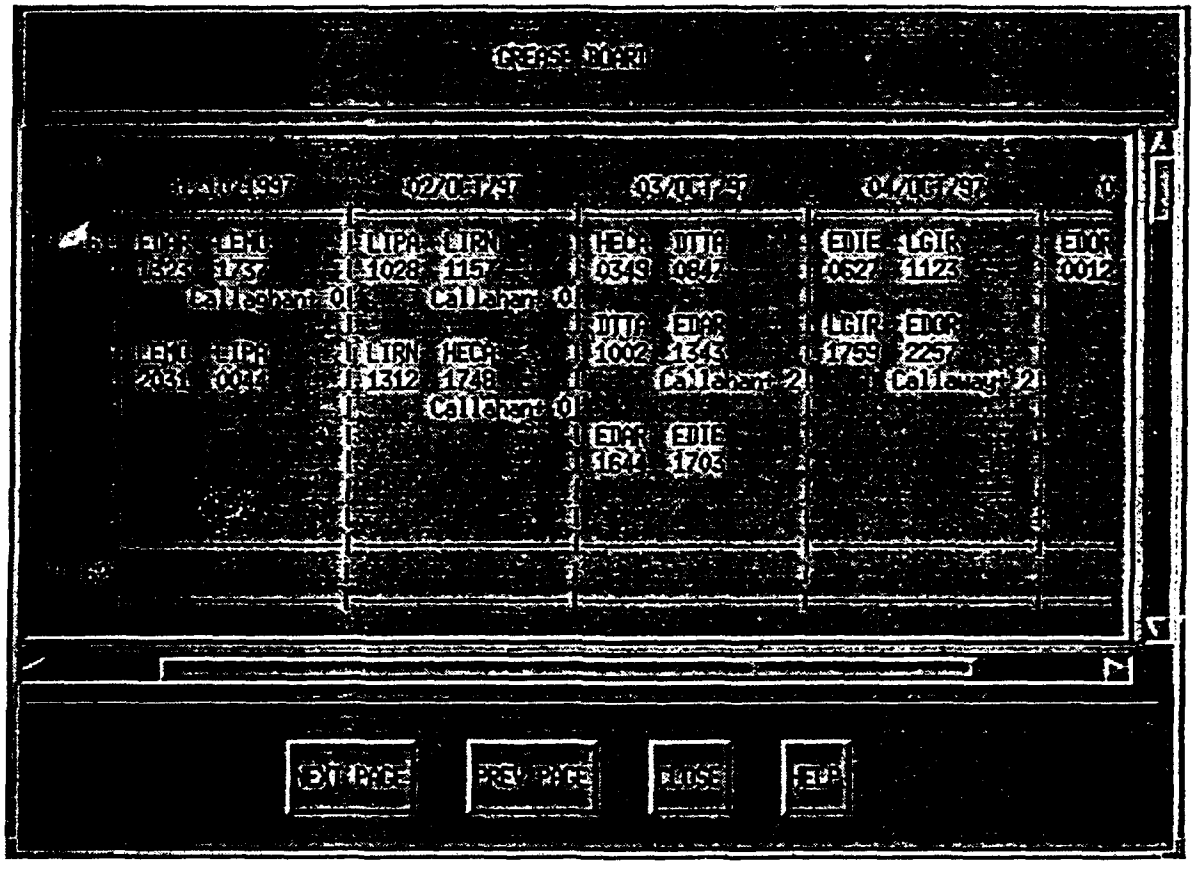

Figure 3.4. The figure shows part of the Greaseboard. The data in the screen can be rolled using the side bars and with the help of the NEXT PAGE and PREV PAGE Buttons.

3.5 *Option Zoom*

The **Zoom** button allows the scheduler to refine the view of the map to show a condensed area of interest or a broader area of interest. After the Zoom button is selected with the left mouse button, several options appear in a pull-down menu, including Zoom-In and Zoom-Out. *Figure 3.5.1* shows the pull-down menu.

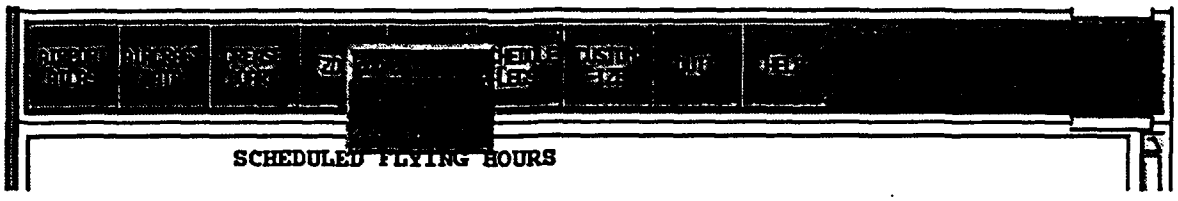

Figure 3.5.1. This figure shows the options available for the changing the look of the map window. Initially only Zoom In is available.

After the right mouse button is used to select the desired option, the cursor on the screen changes to a cross (plus) shape. The cursor is moved within the map window to define a rectangular region -- the left mouse button is used to indicate the upper left corner of the rectangle, and the cursor is dragged until the rectangle is of the desired size, whereupon the left mouse button is released. The map view is updated to reflect the new area of interest. *Figure 3.5.2* shows the zoom process.

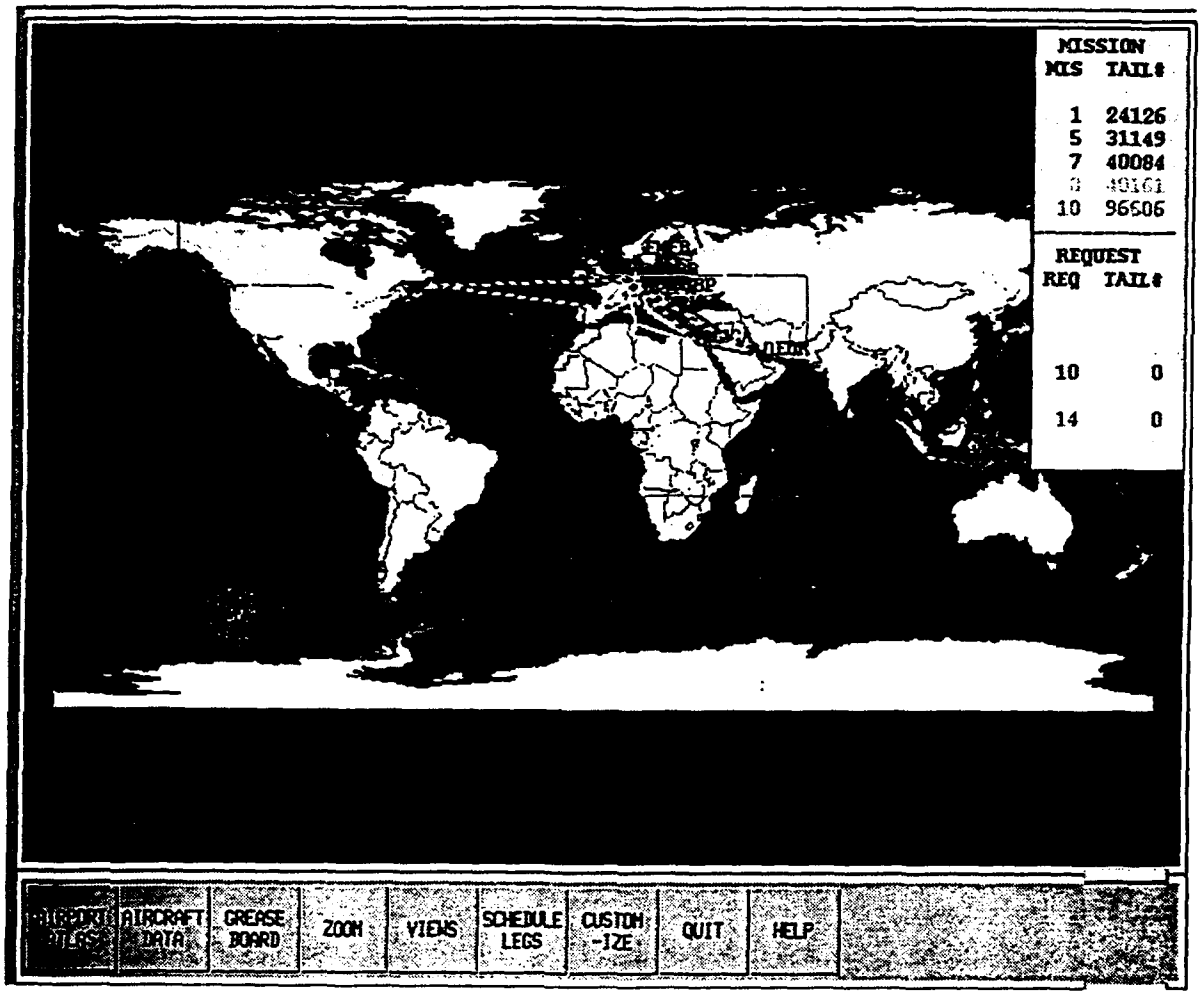

Figure 3.5.2. The figure showing the zooming in progress. Here the cursor is being dragged with the left button pressed. The release of the button indicates the chosen area.

3.6 Option Views

The scheduler can select from 3 possible views -- all plane types, selected plane types, or selected missions. The View button (selected using the left mouse button) produces a pull-down menu with the three options, as shown in *Figure 3.6.1* The left button is used to select the desired option from the resultant pull-down menu.

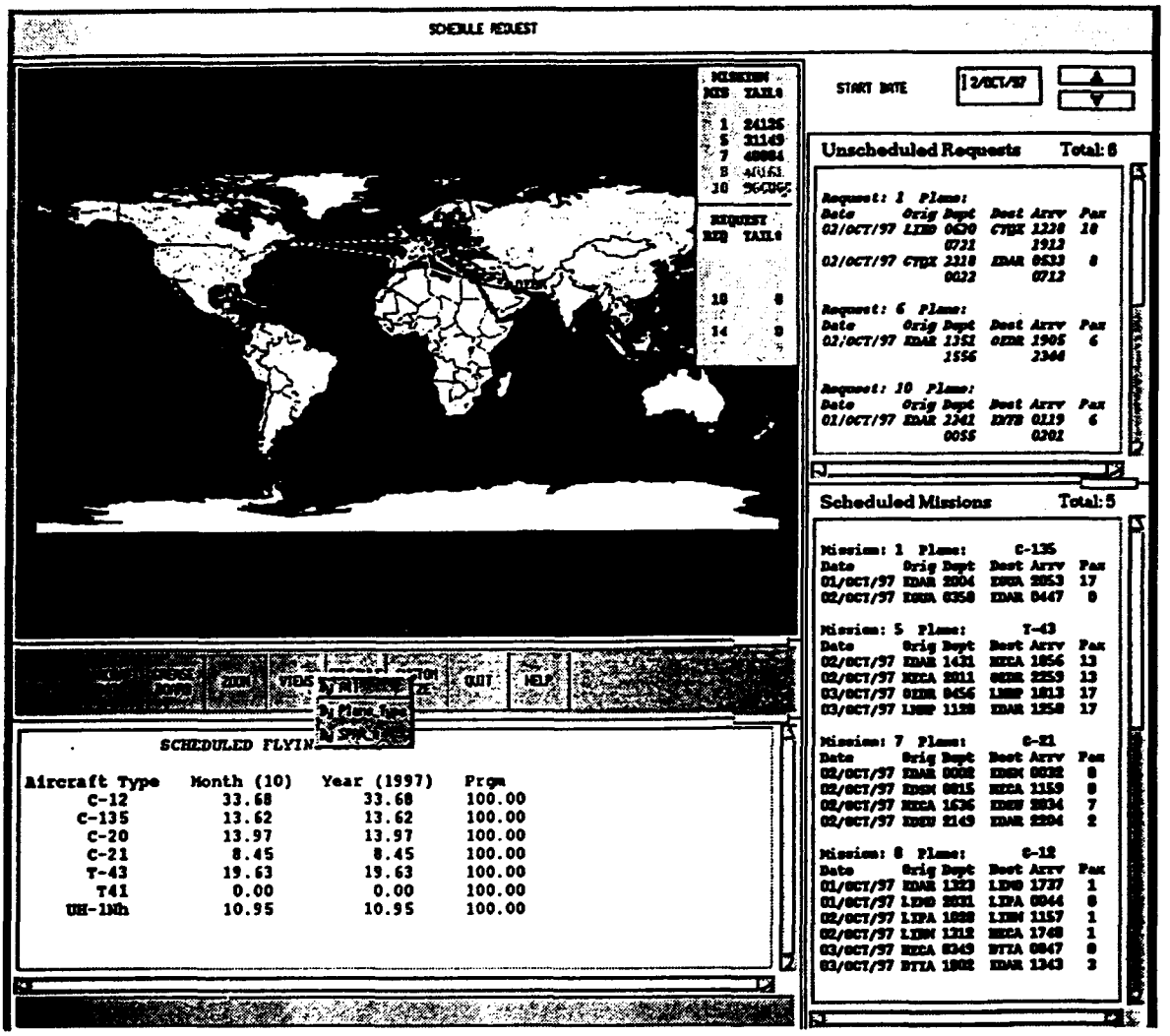

Figure 3.6.1. Pull down from the view button showing three possible views - all plane types, or selected plane types, or selected individual planes.

The desired view is selected using the right mouse button. The default view shows all activity for all plane types when the date is typed in the upper right corner of the screen. If the scheduler is interested in seeing only information about the C-21s and C-20s, the view by plane type is selected, and a pop-up window appears, as shown in *Figure 3.6.2*

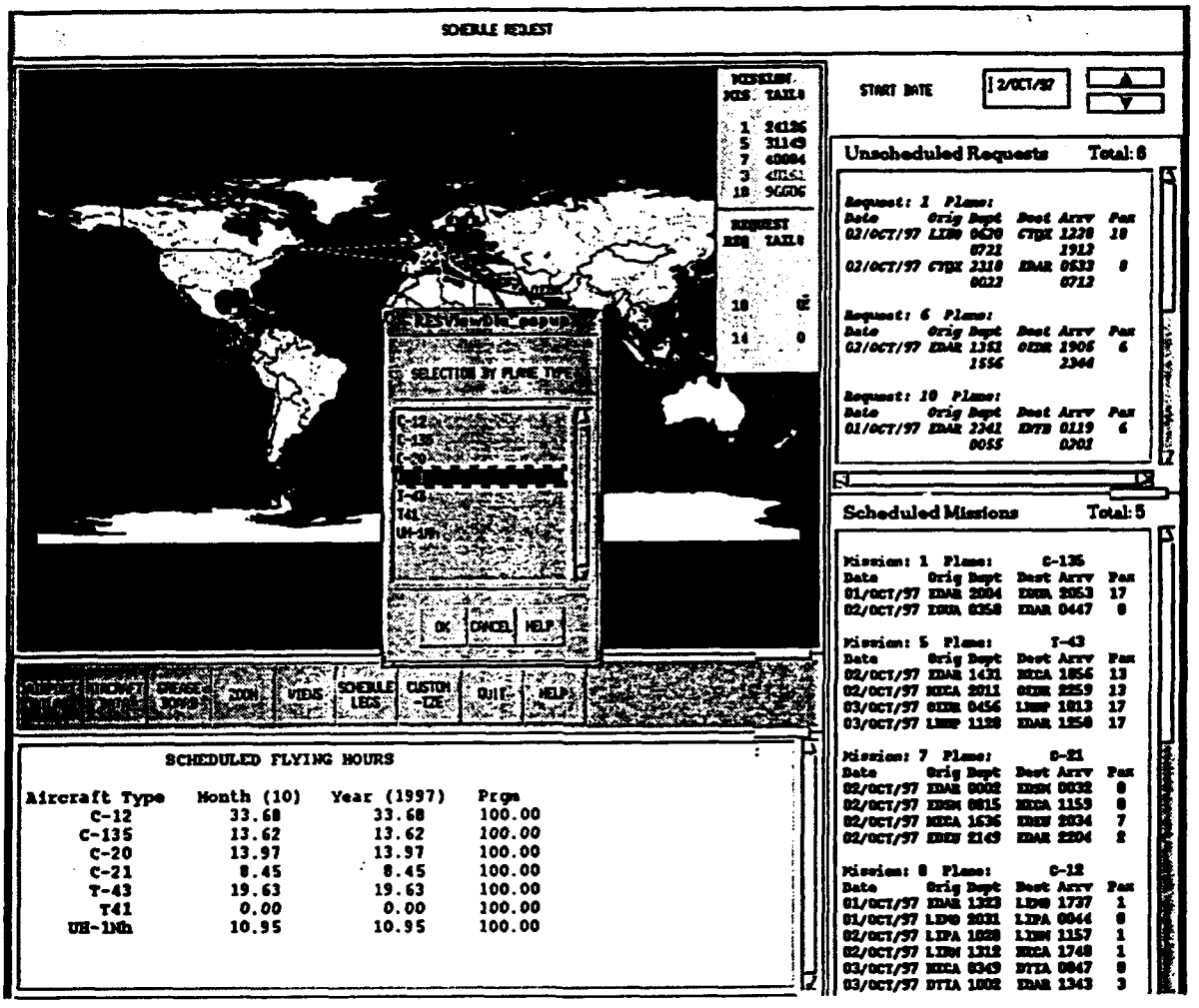

Figure 3.6.2. The pull down menu for selection by plane types. A plane type is highlighted when clicked by the Mouse. A plane type is selected by clicking on the OK button. CANCEL aborts the selection process.

Planes are selected and deselected using a click of the left mouse button. The four areas of the screen are updated to display only information about the selected plane types. View by Spar # pulls up a pop up menu as shown in *figure 3.6.3*

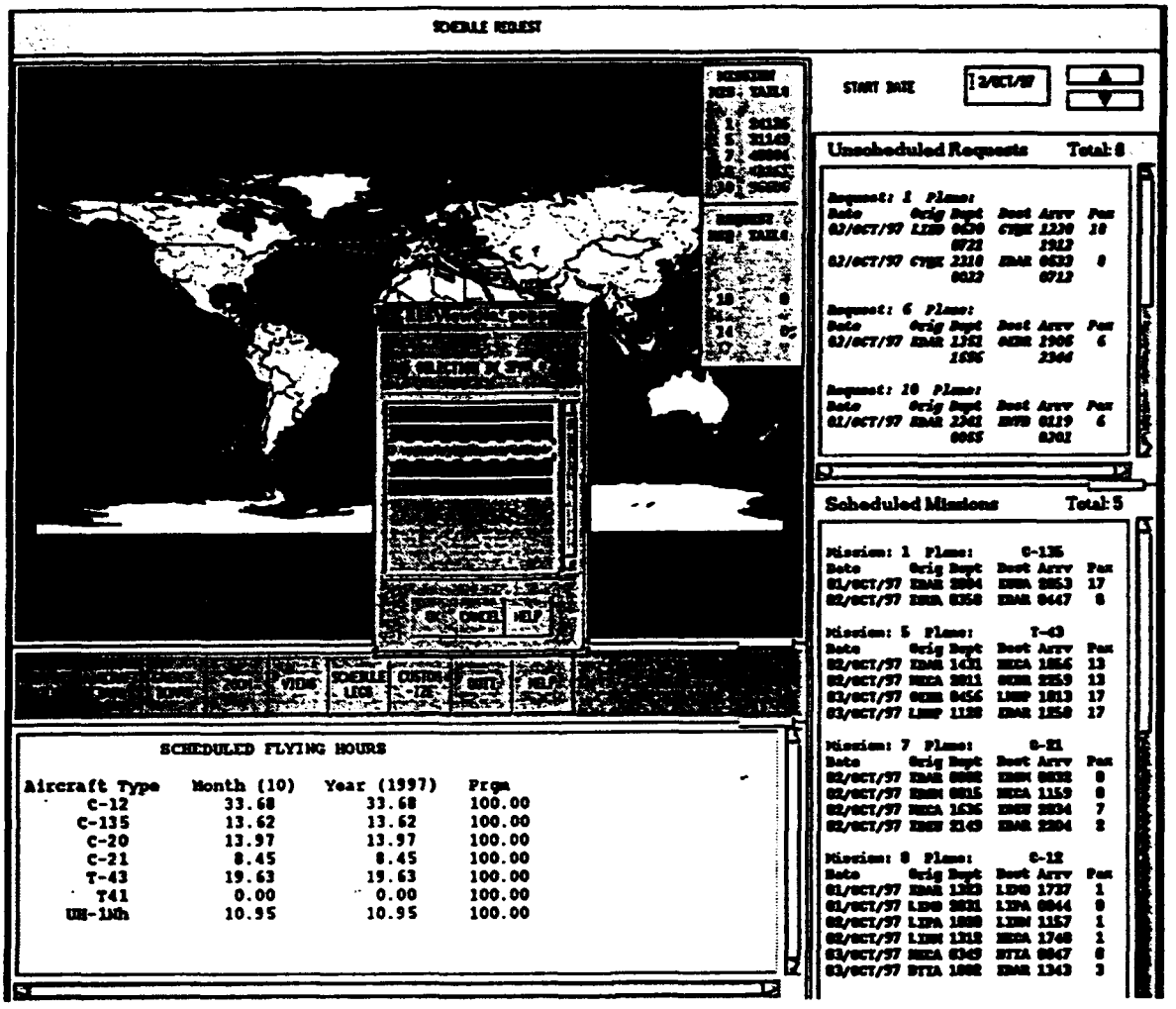

Figure 3.6.3. As shown above the entries in the pop up menu can be selected or deselected by clicking on them. OK confirms the selection and CANCEL aborts it.

The bottom right area of the screen shows flying hour information for two of the views (all plane types or selected plane types). However, if the scheduler chooses to view selected missions, this area displays passenger information for those missions, as shown in *figure 3.6.4*

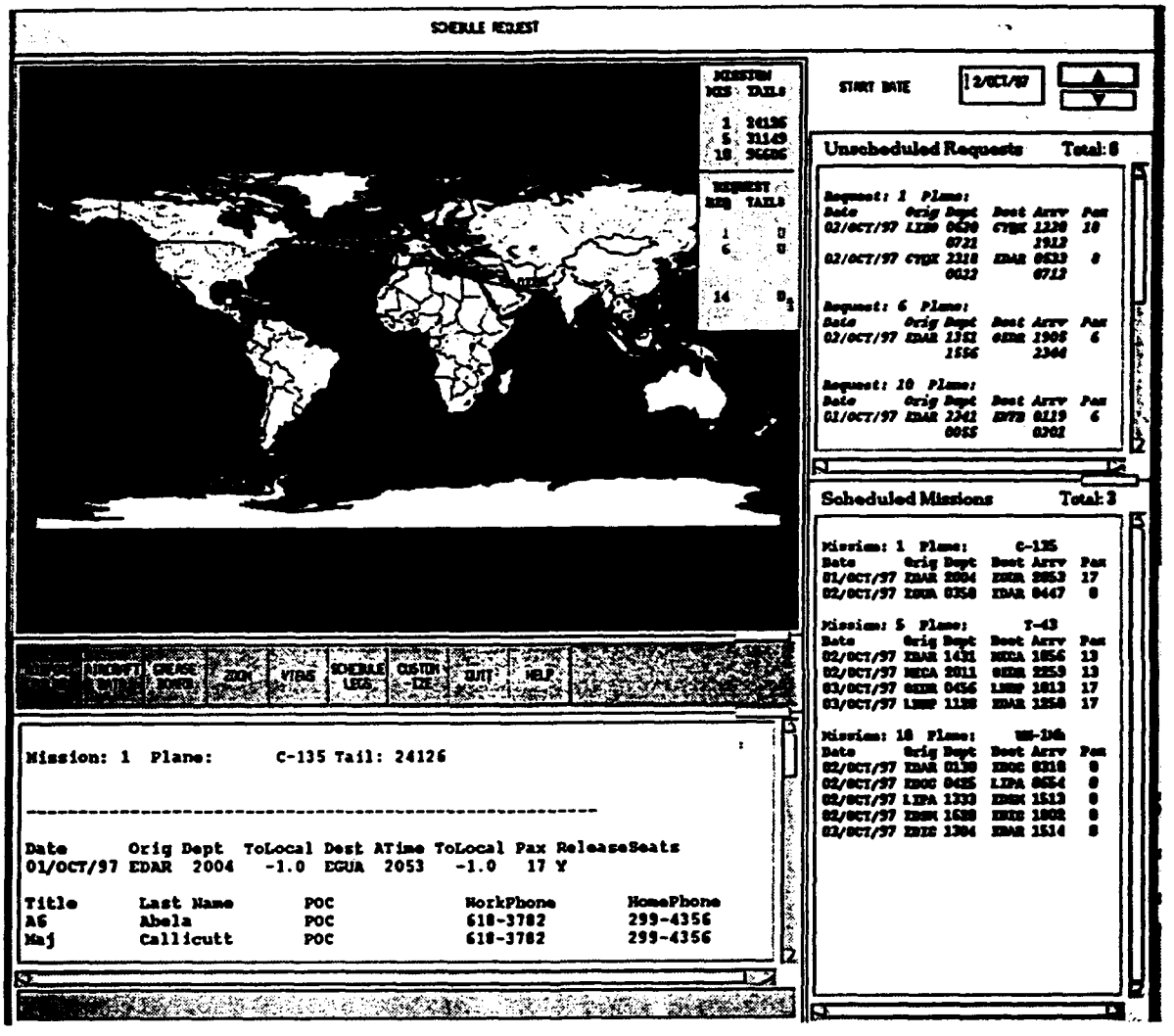

Figure 3.6.4. The selection by Spar # is reflected in the above figure in the bottom left corner. Note that the window has more information on the passengers associated with the mission and the plane type selected.

3.7 *Option Schedule Legs*

The **Schedule Legs** button is used to actually schedule requests by assigning legs of requests to be flown by a particular plane. The process for using this screen is discussed in detail in section III. *Figure 3.7* gives the popup menu for Schedule Leg button.

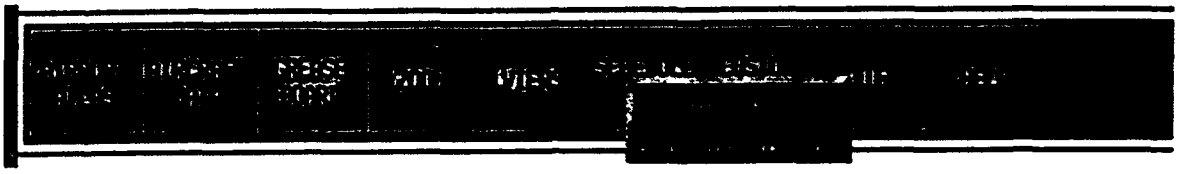

Figure 3.7. The pop up for the Schedule Legs option. Either one can be selected by clicking on the subject. If clicked outside no option is selected.

3.8 Option Customize

The **Customize** button provides options for changing the map display or the mission display. Use the left mouse button to select the Customize option, and choose the appropriate sub option using the right mouse button. The popup menu for Customize is shown in *figure 3.8*

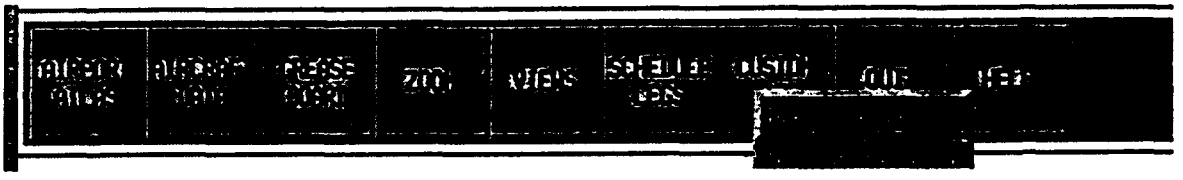

Figure 3.8. The pop up for customizing the screen. Each of these will pull up menus for option on customizing.

The pull down menu for Map Display and Mission Display are shown in *figure 3.8.1.*

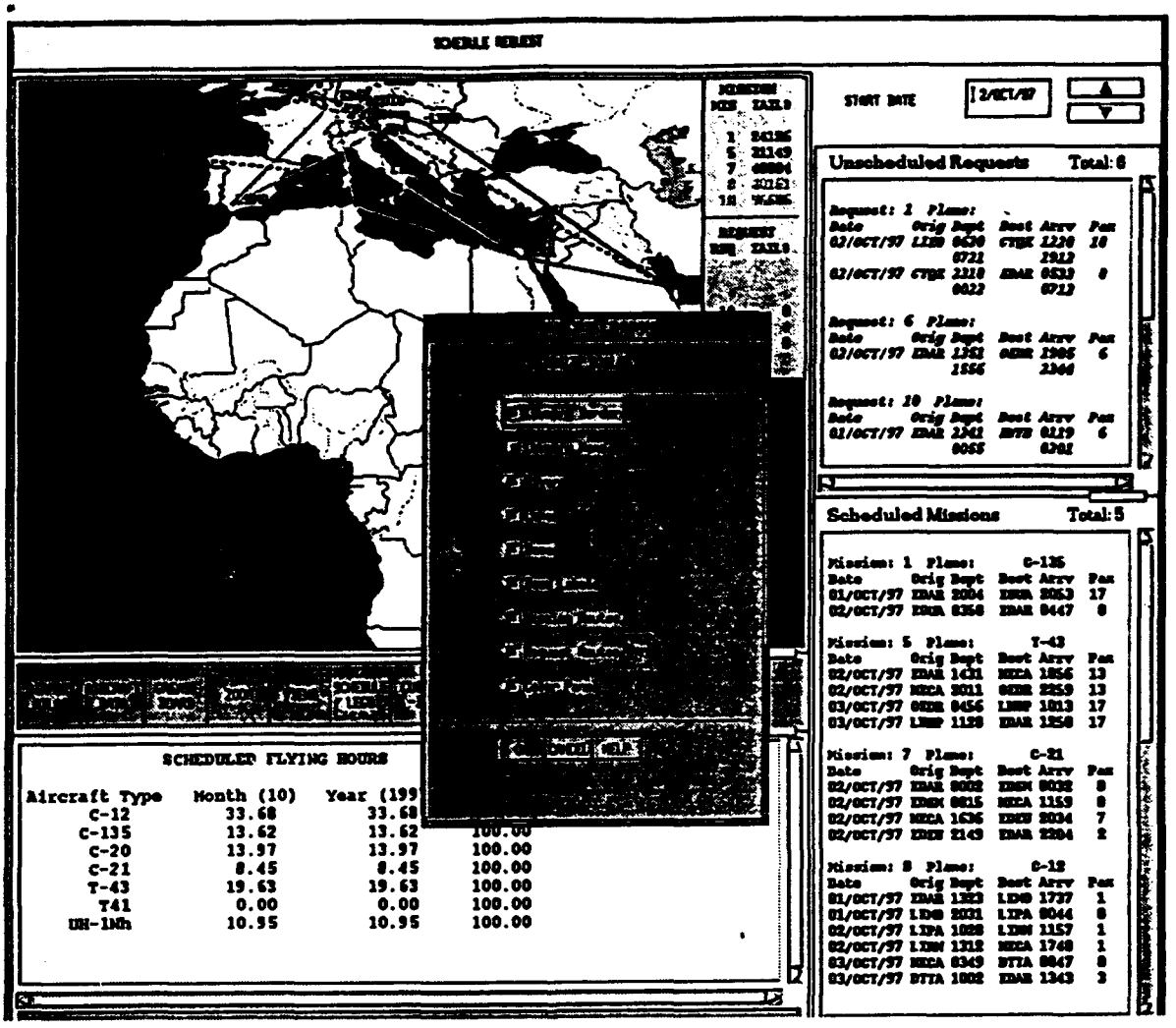

Figure 3.8.1. Customizing the map. Each Option is a toggle. For example, a click of the left mouse button on country border indicates show the country boundaries on the map or don't.

The customizing through the **Missic.. Display** is shown in *figures 3.8.2, 3.8.3* and *3.8.4*. Missions are selected for customizing through the pop up menu as shown in *figure 3.8.3*. The state of the map window before and after the customization is shown in *figure 3.8.2* and *figure 3.8.4* respectively.

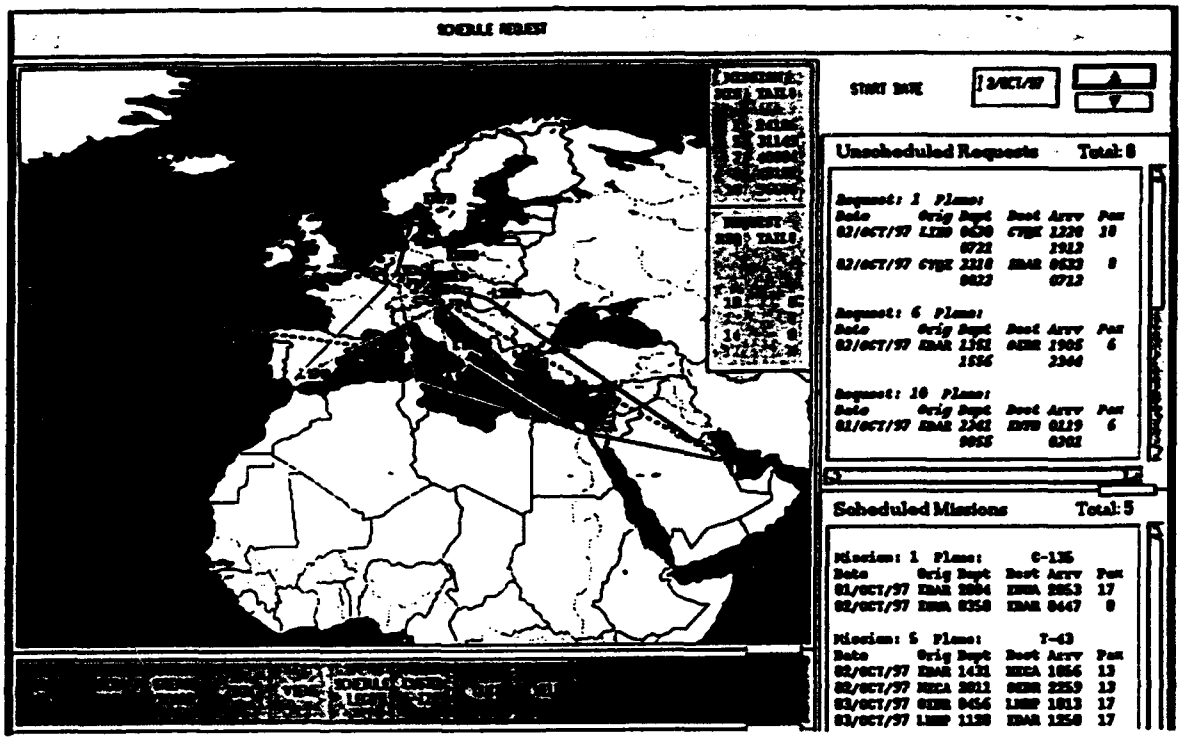

Figure 3.8.2. A in instance of the map with zoomed in appearance. This is the appearance before the customize by mission display is called.

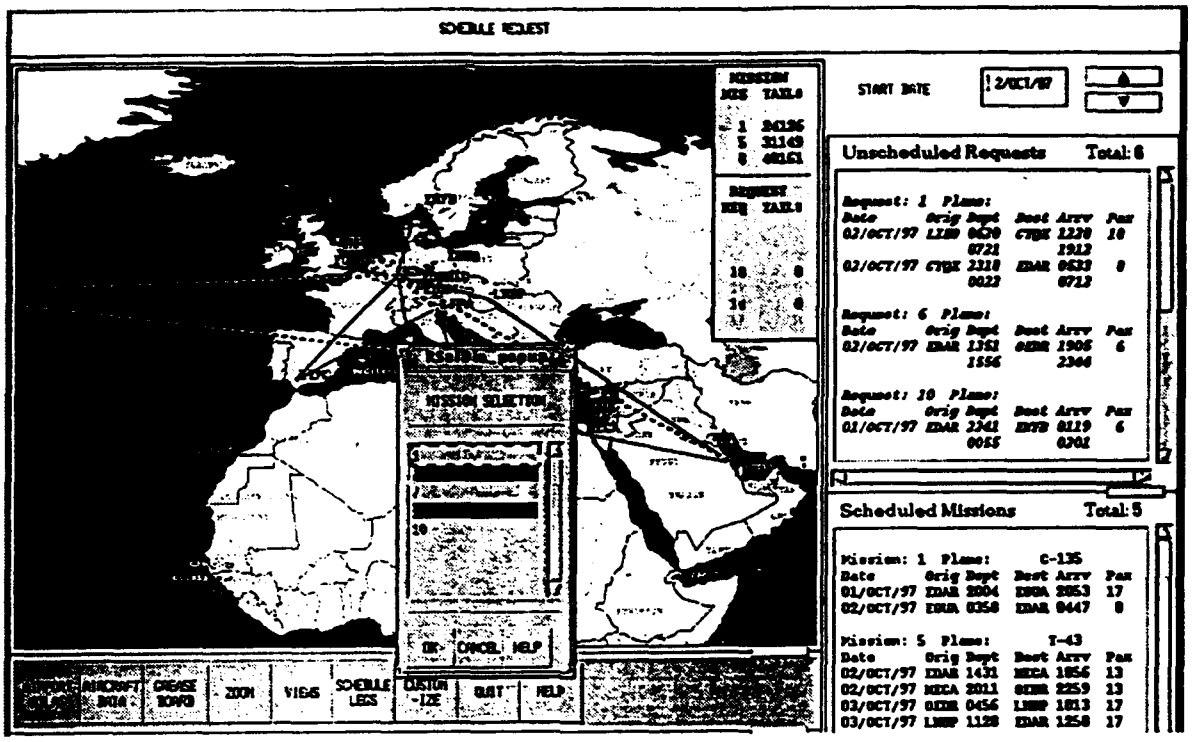

Figure 3.8.3. Customize option for mission display is shown. In the figure missions 1, 7 and 10 are being selected.

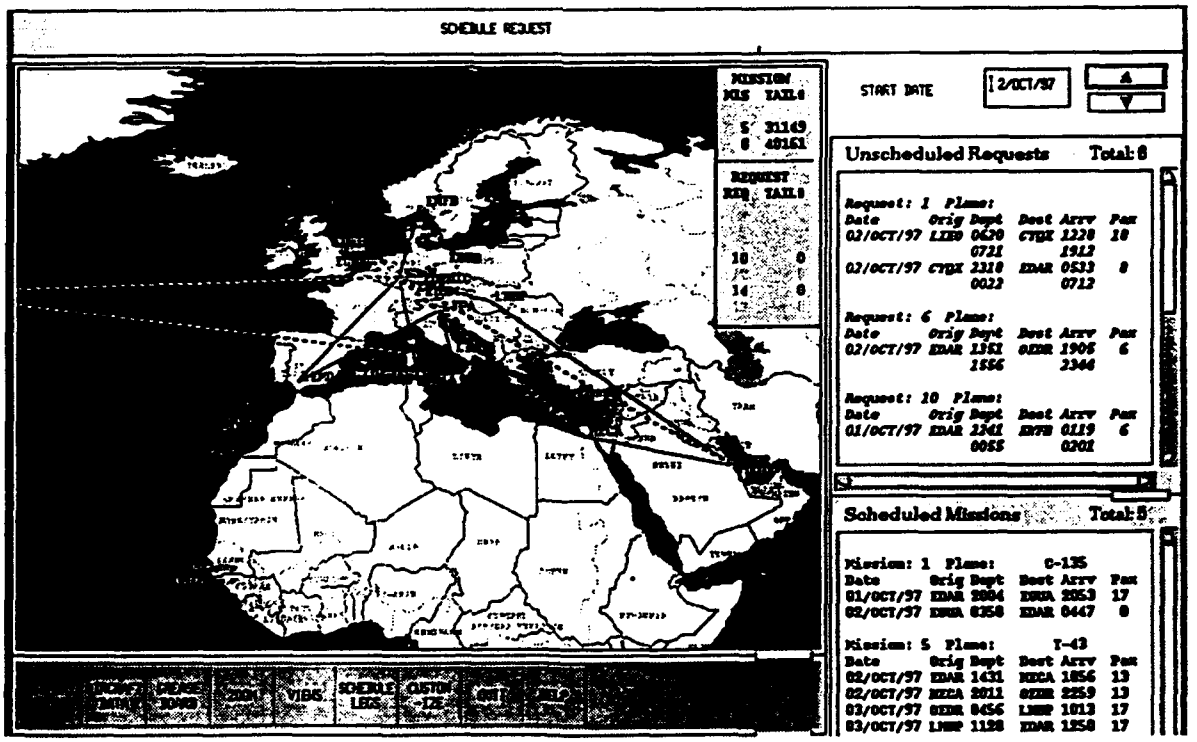

Figure 3.8.4. The appearance of the map after customizing.

IV. SCHEDULE LEGS WINDOW

The Scheduling window helps you schedule legs for the mission. *Figure 4* shows the screen for the scheduling window. In all the menus at any level the selection of the *HELP* option will display help regarding a particular screen. The menu at the top gives the options of *FILE, DEFINITION, RUN* or *HELP*. The upper half of the screen displays the missions while the lower half displays all the requests associated with a mission.

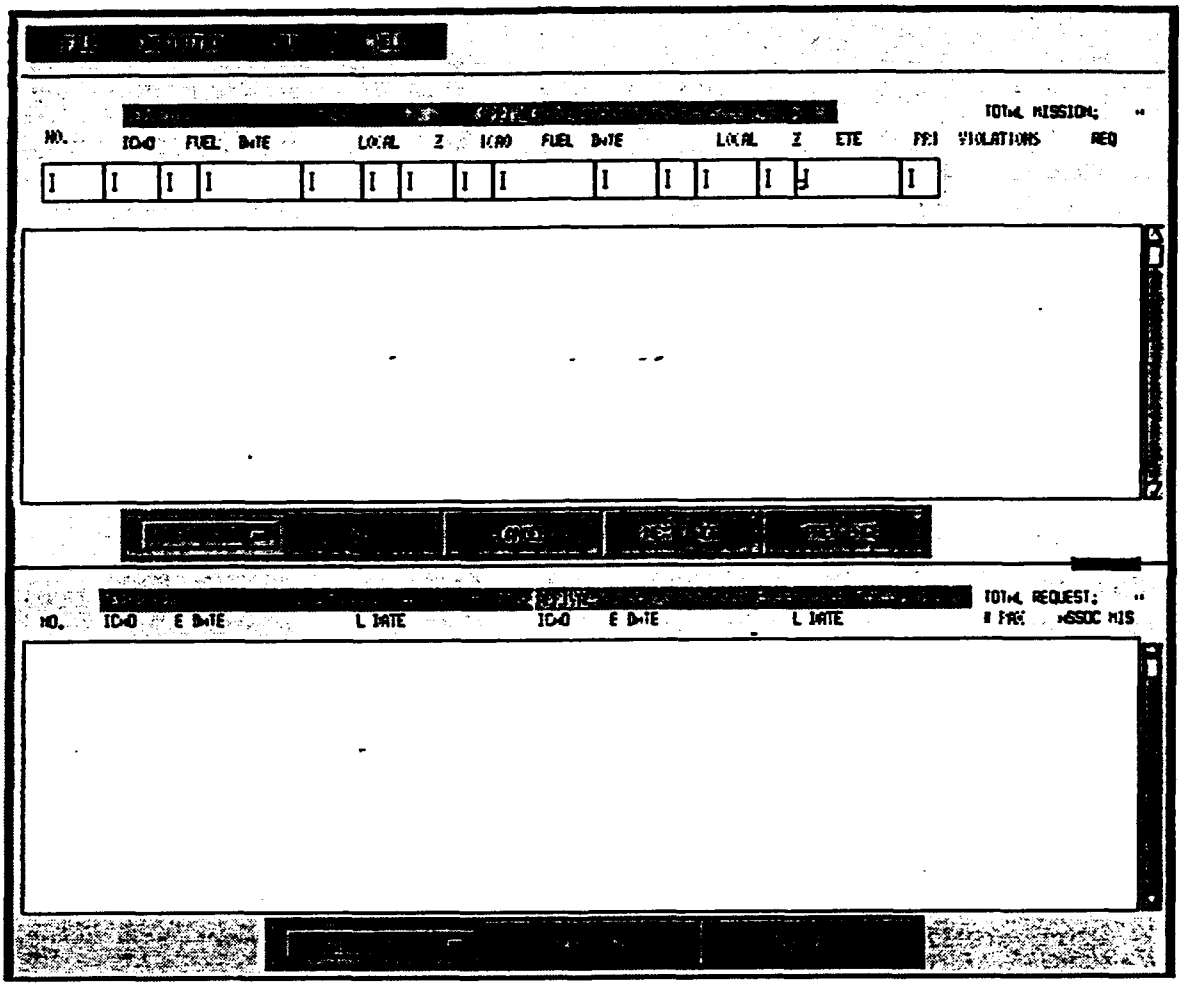

Figure 4. The pull down of the schedule legs screen. The options in the top left corner assist in the scheduling.

4.1 Option File

To select the options related to the file move the cursor to the FILE and click the first button. Another pop up window appears shown in *figure 4.1.1.* This option facilitates the use of files to store or retrieve data, and make the changes permanent. The options are described in more details below.

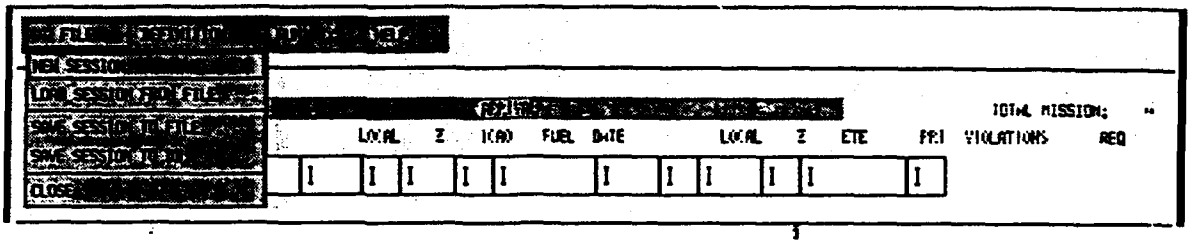

Figure 4.1.1. The pop up menu for the option FILE.

NEW SESSION: The *NEW SESSION* indicates that the work done will be fresh from start and bears the appearance of a clear screen.

LOAD SESSION FROM FILE : If there was a session that was earlier stored in a file, the session can be retrieved from the option *LOAD SESSION FROM FILE.* This will ask for the file name where the session was stored.

SAVE SESSION TO FILE: It is sometimes a good idea to store a session to a file towards the end of a session if the scheduler is not finished with the process or is unwilling to make the changes permanent. The option *SAVE SESSION TO FILE* asks for a file name where the file is to be stored. Both these options pop up a window shown in *figure 4.1.2.* The pop up menu has four options. The **OK** is used if the entries are correct and final. The FILTER allows the user to edit the path name or file. The selections can be terminated by clicking at the Cancel option.

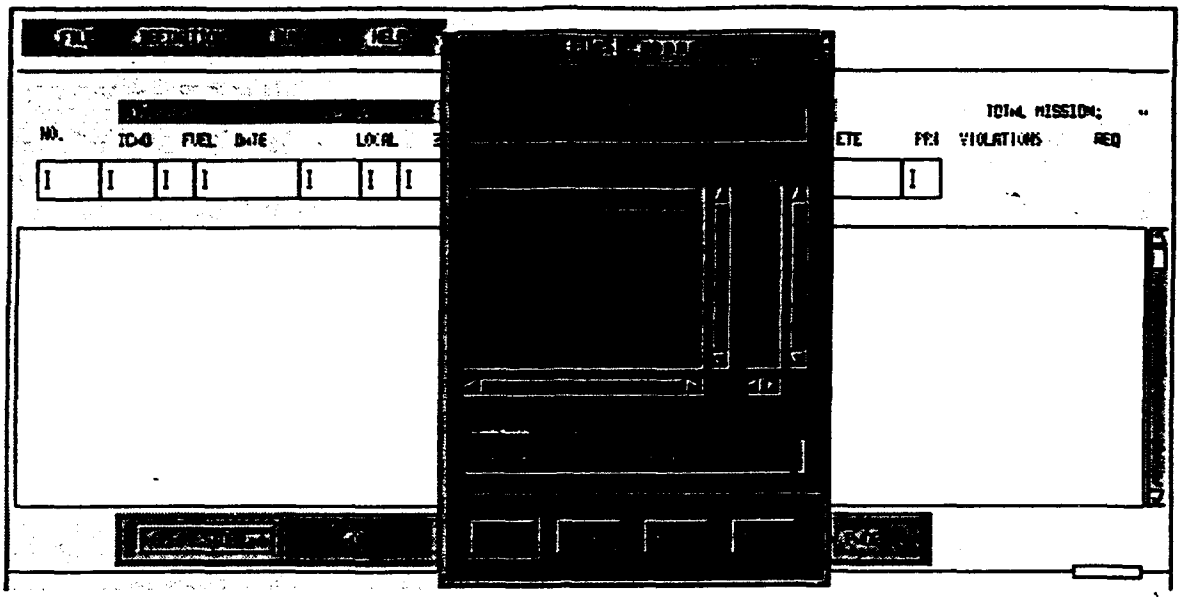

Figure 4.1.2. The pull down menu to save or load a session from or to a file.

SAVE SESSION TO DB : This option causes the changes to be permanently stored to the database.

CLOSE : Scheduling Screen can be exited with this option.

4.2 Option Definition :

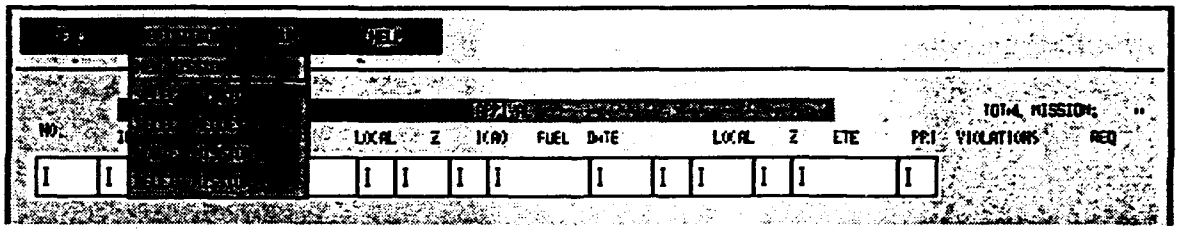

Figure 4.2.1. The pull down menu for DEFINITION for mission.

NEW MISSION : Selected to create a new mission. The selection of this option will pull up another menu as shown in *figure 4.2.2*. The mission ID is a serial field used by the database and cannot be altered by the user. The Tab key is used to move between fields. The tab takes the cursor to the next field. The data is entered as shown is *figure 4.2.3*. The data can be made available for acceptance by clicking on the OK Button. The Cancel simply ignores the entry and returns to the previous screen. It should be noted however, that the session is NOT written to the database unless the Save Session to the Database is selected from the file menu.

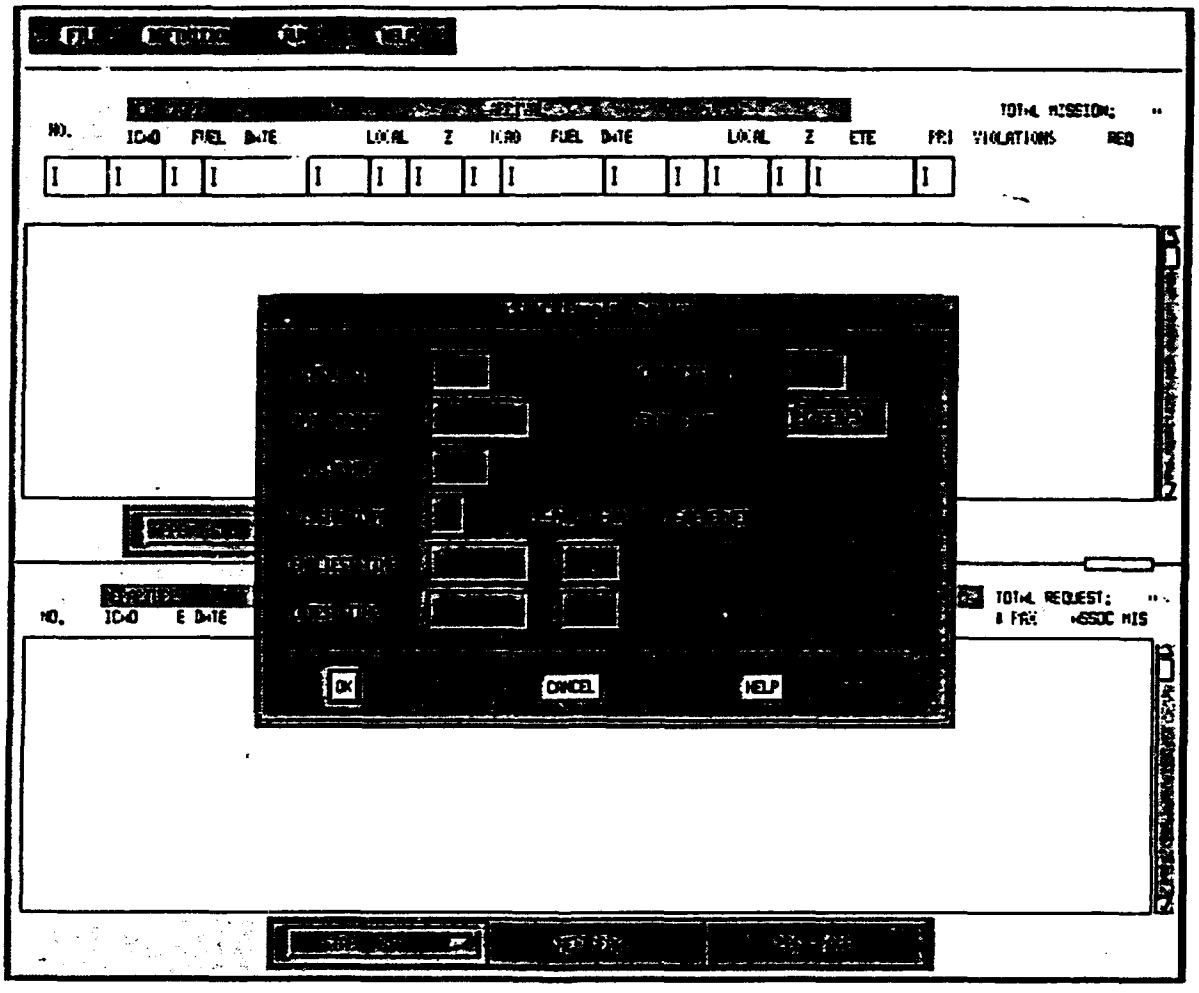

Figure 4.2.2. The pop up menu for Creating a new mission. The Mission ID is created by the Database for a valid entry.

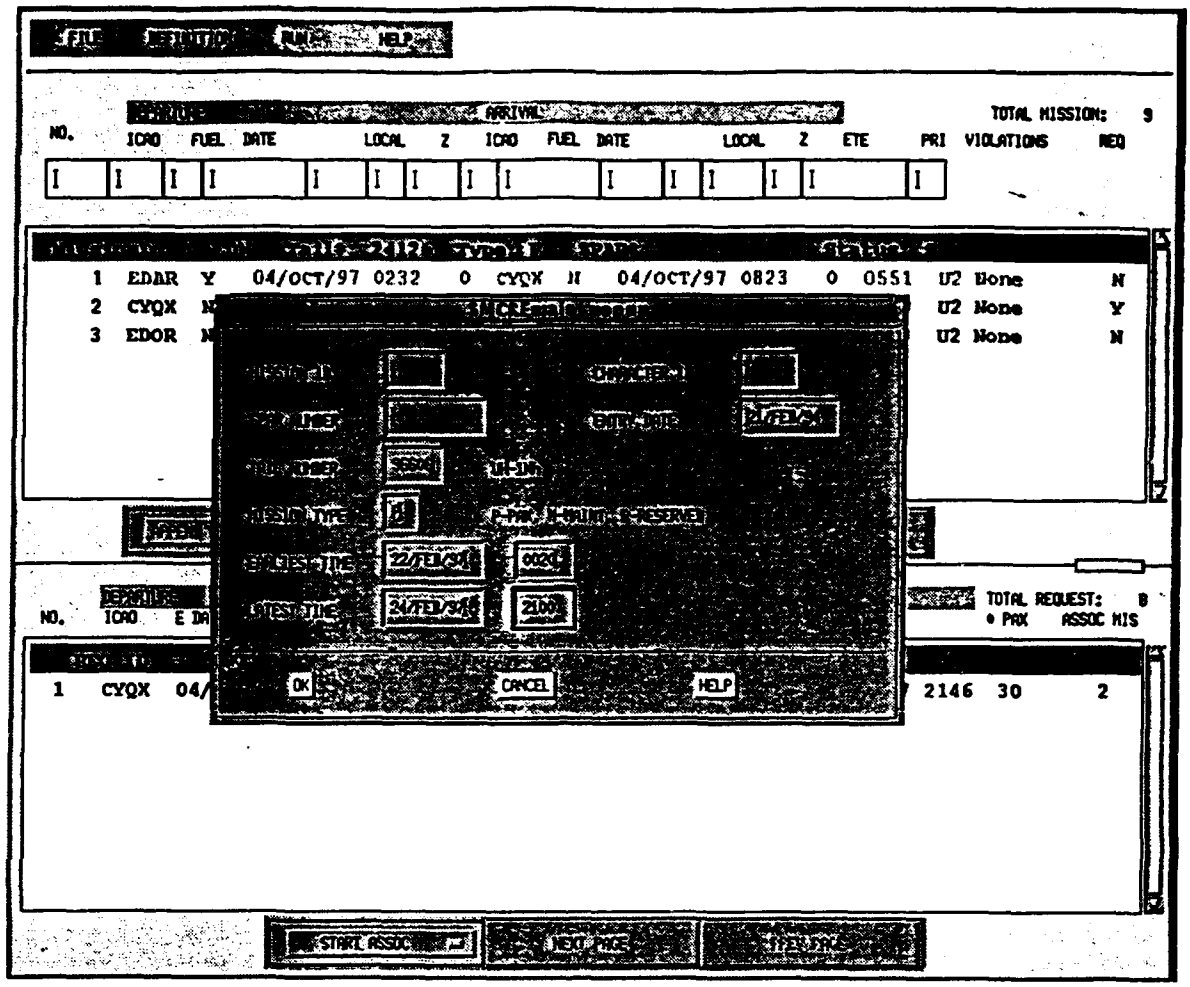

Figure 4.2.3. An example of the previous pop up menu with details in the fields.

SELECT MISSION : This option facilitates in retrieving an already existing mission from the database. This option is selected as shown in *figure 4.2.4.*

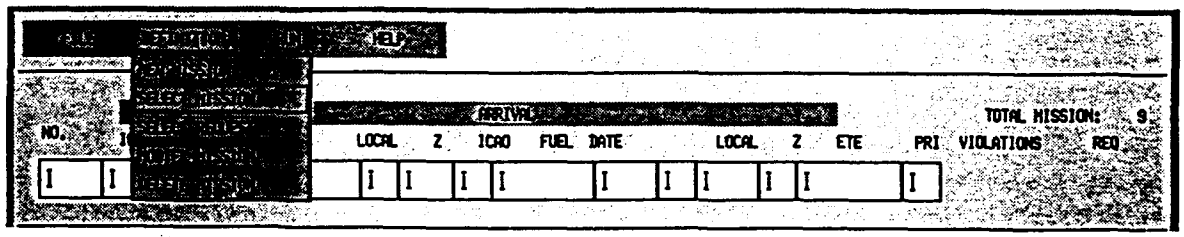

Figure 4.2.4 To Select an existing mission. Click as shown in figure.

SELECT MISSION pulls up a menu as shown in figure 4.2.5.

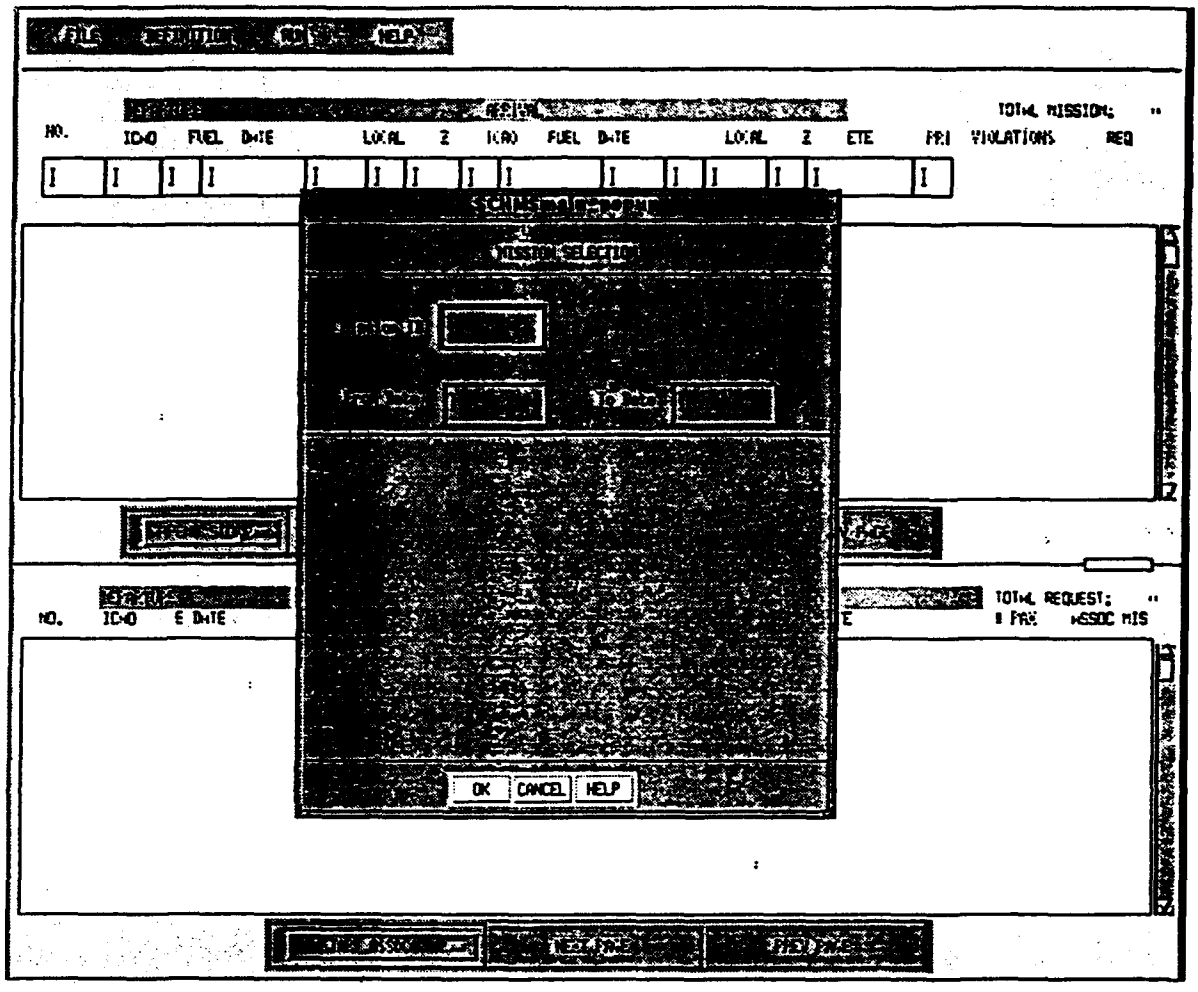

Figure 4.2.5 The pop up for SELECT MISSION.

It is desirable to enter the mission ID number. If the other fields are left unfilled, the record matching the mission ID will be selected. An example is shown in *figure 4.2.6.1* Since Mission ID is unique for each mission, only one record is selected. Alternatively if all the missions between two dates are desired then Mission ID field can be left unfilled and the From Date and To Date are entered. This will bring up information about all the missions between the 2 dates as shown in *figure 4.2.6.2*. Of these missions one of the missions is selected by highlighting it and clicking on the OK button. If no missions are to be selected CANCEL aborts the selection.

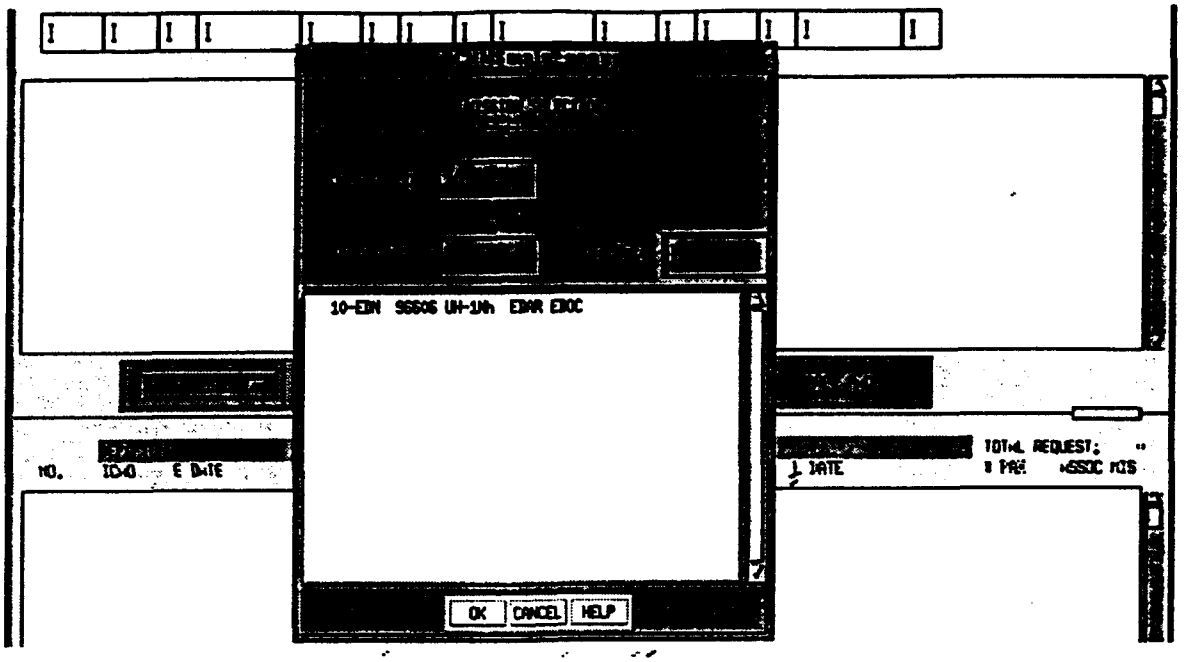

Figure 4.2.6.1 An example of selecting mission with Mission ID 10. This figure gives a short description of the mission.

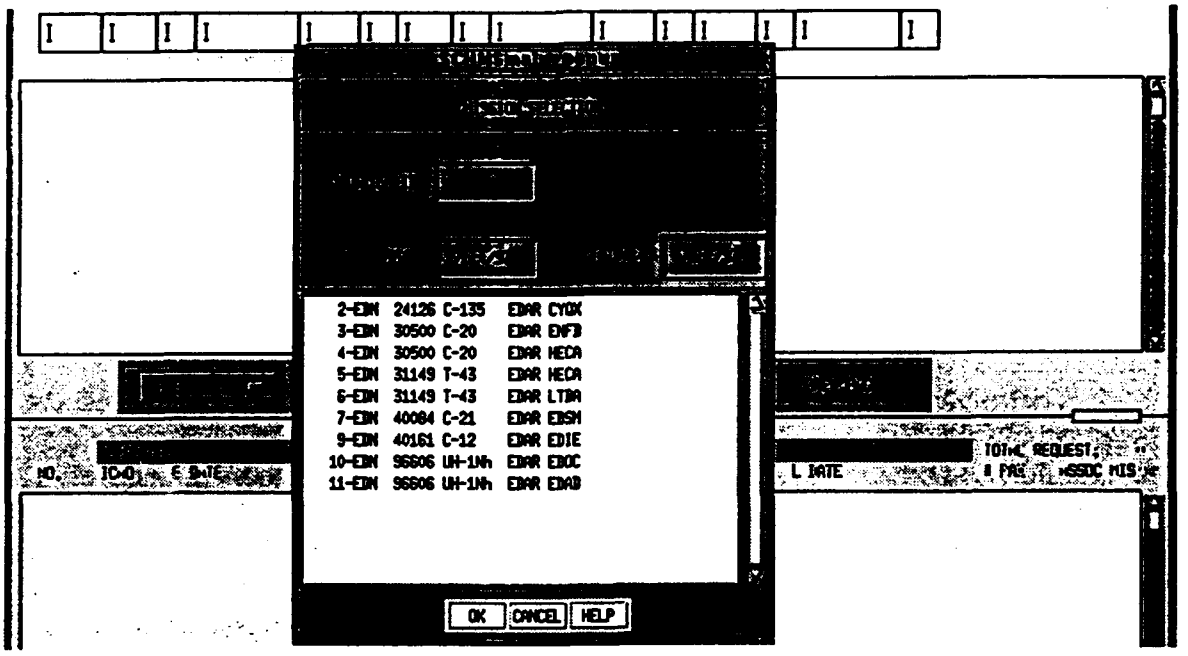

Figure 4.2.6.2 An example of displaying all the missions between the dates October 2, 97 and October 5, 97. Clicking on any one of them and OK will select that mission into the scheduling screen.

The description(s) of the mission for the corresponding entries are displayed in the dialogue box. To select this mission simply click on the OK button. To cancel click on the CANCEL button. If the OK is selected the details of the mission are retrieved from the database and displayed on the screen. The upper half is displays the missions with the particular mission ID and the lower half with the requests associated with the corresponding mission ID. *Figure 4.2.7* displays one such entry. In the example Mission ID 10 has five legs. The entries for Tail number, Type, Spar # and Status are shown in the form of a header. The details show Departure details (ICAO code, Fuel Stop(Yes/No), Local Date and Time, Z) and ARRIVAL details(ICAO code, Fuel Stop, Local Date and Time, Z, ETE, Priority, Violations and Request Code). A Y in the REQ field signifies that there is a request associated with the particular Mission leg and N indicates no request is associated.

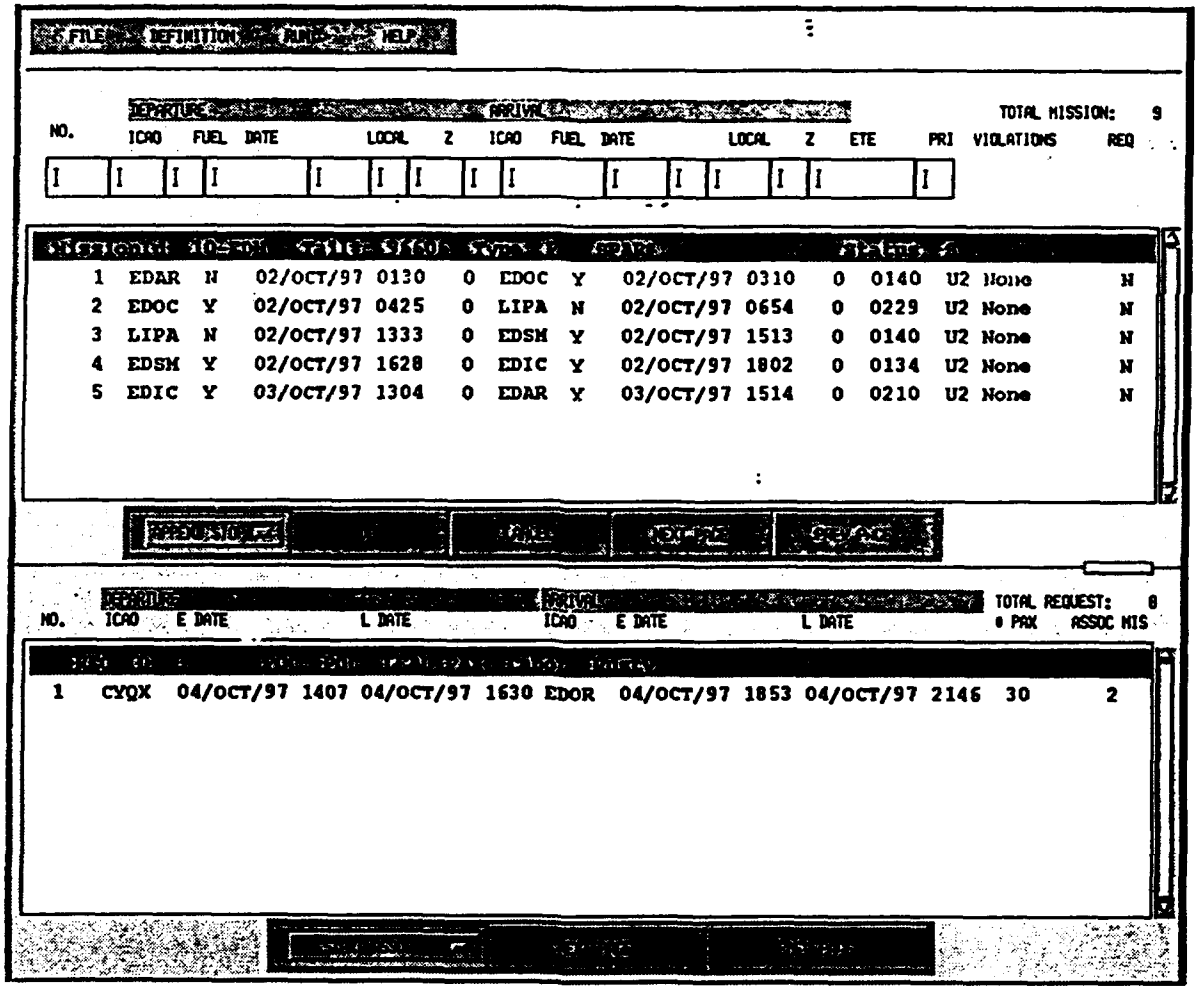

Figure 4.2.7. Selecting all missions between October 2, 97 and October 5, 97. The details of various Requests are displayed in the lower half of the window. In the figure, the button NEXT PAGE in the highlighted state indicates that more than one screenful of data is available.

The leftmost button in the upper half of the screen allows the following processes. The pop up menu is as shown in *figure 4.2.8.*

- APPEND STOP : To append a stop in a mission leg.
- MODIFY LEG : To modify a mission leg.
- REM O-STOP : Remove a origin Stop
- REM D-STOP : Remove a Departure Stop.
- NO EDIT : Do not allow edit.

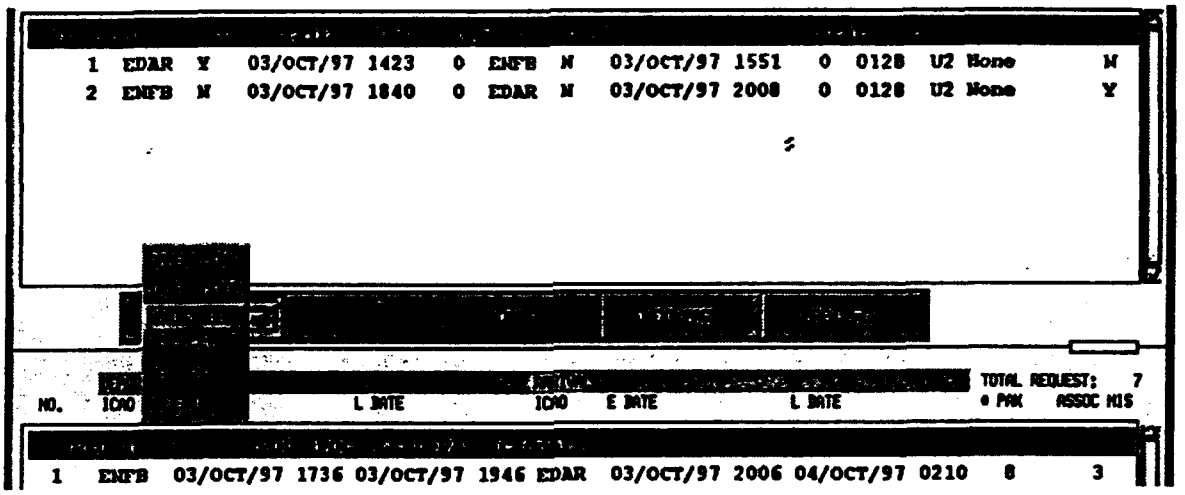

Figure 4.2.8. The pop up menu for the STOP.

To work with the above options a mission leg should be selected. The selection is done by moving the cursor over to the displayed record and clicking with the left mouse button. Highlighted row is the indicator of a leg being selected. A mission leg can be deselected by clicking on a highlighted row (clicking on a selected leg). The entries get shown in the boxes above the upper window. At this point the a stop can be added removed or the mission leg can be modified using the options shown in *figure 4.2.8.* Once completed the changes are confirmed by selecting the OK button. CANCEL aborts the changes. The entry is checked for violations.

If the mission legs detail is more than one screen then NEXT PAGE and PREV PAGE are used to browse through the screens. It should be noted that NEXT PAGE and PREV PAGE options work if there are data available in the previous and next page respectively.

SELECT REQUEST : An existing request can be selected through this option of the definition button as shown in *figure 4.2.9.*

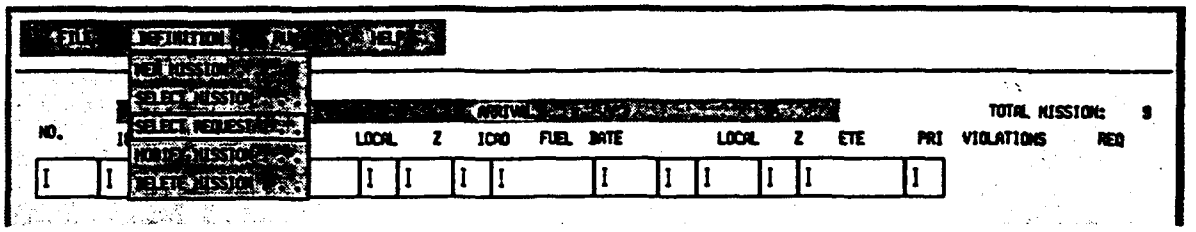

Figure 4.2.9. The position of the cursor for Selecting an existing Request.

This pulls up another pop up window as shown in *figure 4.2.10.* The existing requests are displayed and more requests can be selected by entering the Request ID, name of the lead passenger or the point of contact. Double click on the requests deselects them. The displayed requests are then selected if OK button is clicked. CANCEL aborts the Request Selection process.

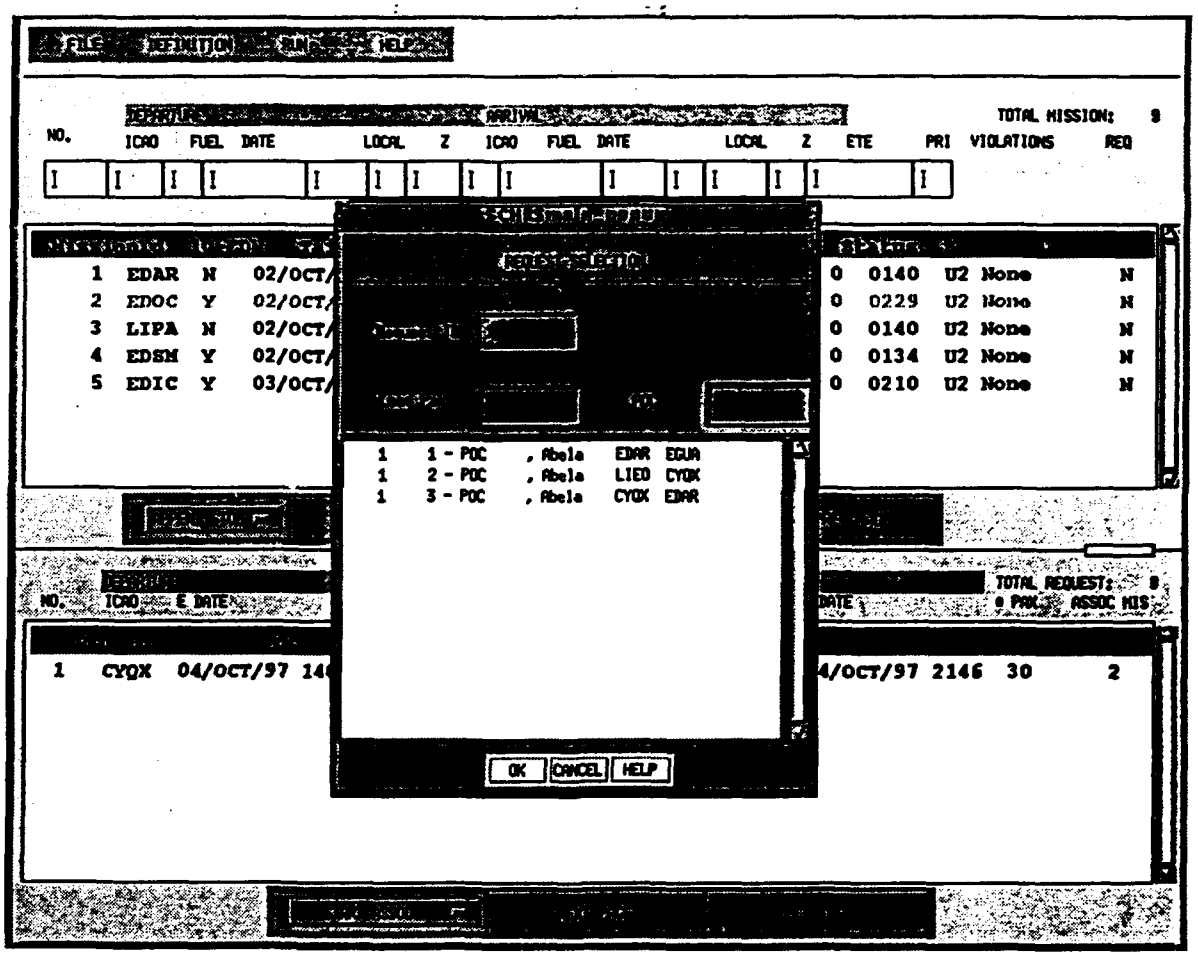

Figure 4.2.10. The pop up window for Se g a Request. The figure above signifies that three requests were found with Request ID 1.

MODIFY MISSION : This option allows an existing mission to be modified. *Figure 4.2.11* shows the selection of this option from the DEFINITION.

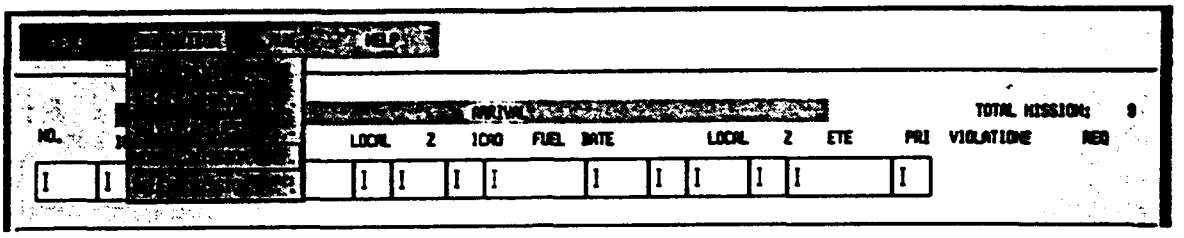

Figure 4.2.11. Selecting MODIFY MISSION option from the DEFINITION.

The popup menu that is pulled out is shown in *figure 4.2.12*. Since the mission ID can not be changed this field is shown with a lower typeface. The other entries can be modified. The changes may be accepted or ignored through the OK/CANCEL buttons respectively. If OK is selected the new details are stored under the same Request ID.

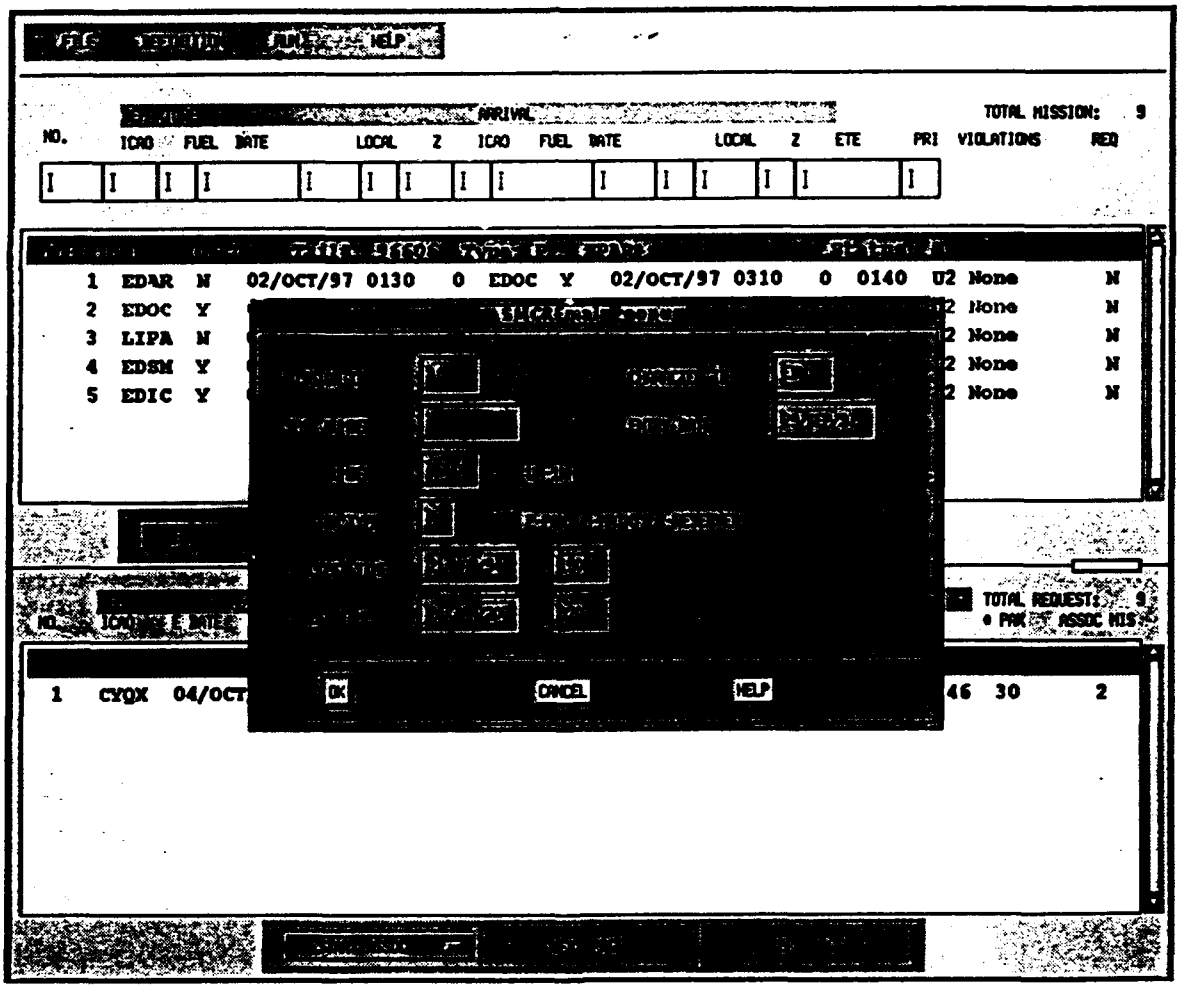

Figure 4.2.12. The window to enter the modifications in a mission.

DELETE MISSION : To delete a mission click on the option DELETE MISSION from the FILE menu as shown in *figure 4.2.13.*

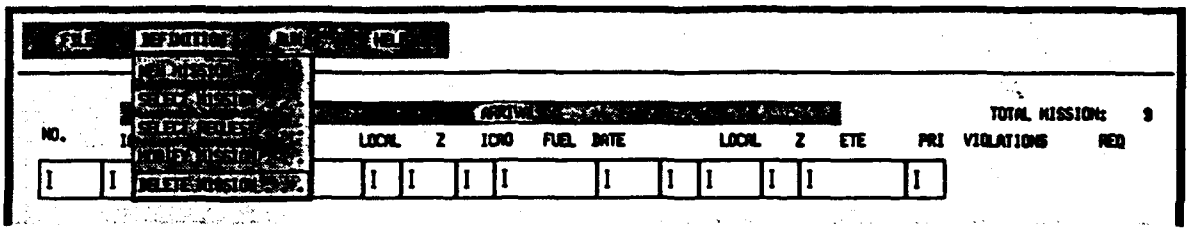

Figure 4.2.13. Selection of DELETE MISSION from DEFINITION.

This option deletes the existing mission being displayed in the upper window. The popup menu is similar to the one shown in *figure 4.2.12.* If no missions are in the window, the user is asked to supply the details. In case a mission already exists, this mission will be prompted for deletion. The details of the existing mission are shown in a lighter typeface to indicate that this can not be altered. The option should be confirmed through OK or canceled through CANCEL.

4.3 Option Run :

The scheduling can be done either manually or through the built-in optimization algorithm. In either case it is required that a mission be selected.

Manual Scheduling :

The manual scheduling describes the scheduling of requests with missions based on manual observation and decision. The following sections describe the process of unscheduling and scheduling of requests with explanation of the column entries. For the ease of explanation we will call the upper half of window Mission widow as it shows the details of a mission with different legs. The lower half will be termed as Request Window as it shows the requests for a particular Request ID. To begin scheduling it is necessary that at least one mission has been selected.

4.3.1 UNSCHEDULING A REQUEST

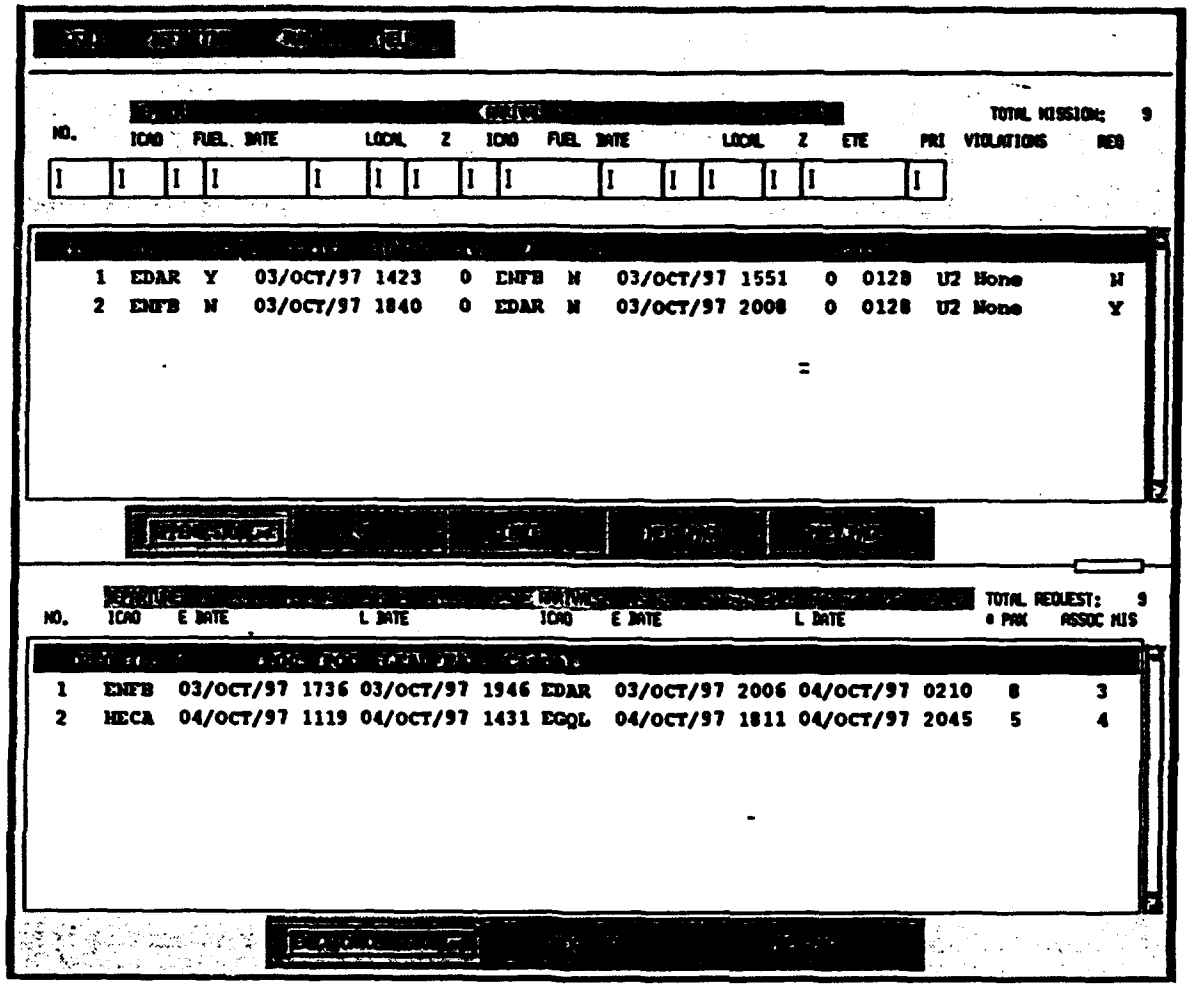

Figure 4.3.1. Mission 3 with 2 legs. Leg 1 has no request associated with it (N in the REQ column). Leg 2 has request associated wit it (Y in REQ column). The entries 3 and 4 in the ASSOC MIS column of the Request screen indicates that the requests are associated with Missions 3 and 4 respectively.

Figure 4.3.1 shows the scheduling screen with details of Mission 3 having 2 legs. As described in the Schedule legs section, the **REQ** column has either a N or Y entry. The N for the first leg in the mentioned figure indicates that it has no scheduled request. While the Y for the second leg indicates that the leg has a scheduled request. The Request window in the same figure has two entries denoting that there are two requests. The last column, termed **ASSOC MIS**, indicates the mission with which the request is associated. In the figure shown below, request 1 is associated with mission 3 while request 2 is associated with Mission 4. We will unschedule the first request.

To start scheduling click on the leftmost button at the bottom of the screen. The cursor takes the shape of a pointed finger. Click on the first request. The request gets highlighted. At the same time the legs with which this request is associated gets highlighted in the Mission window. This change is shown in *figure 4.3.2.*

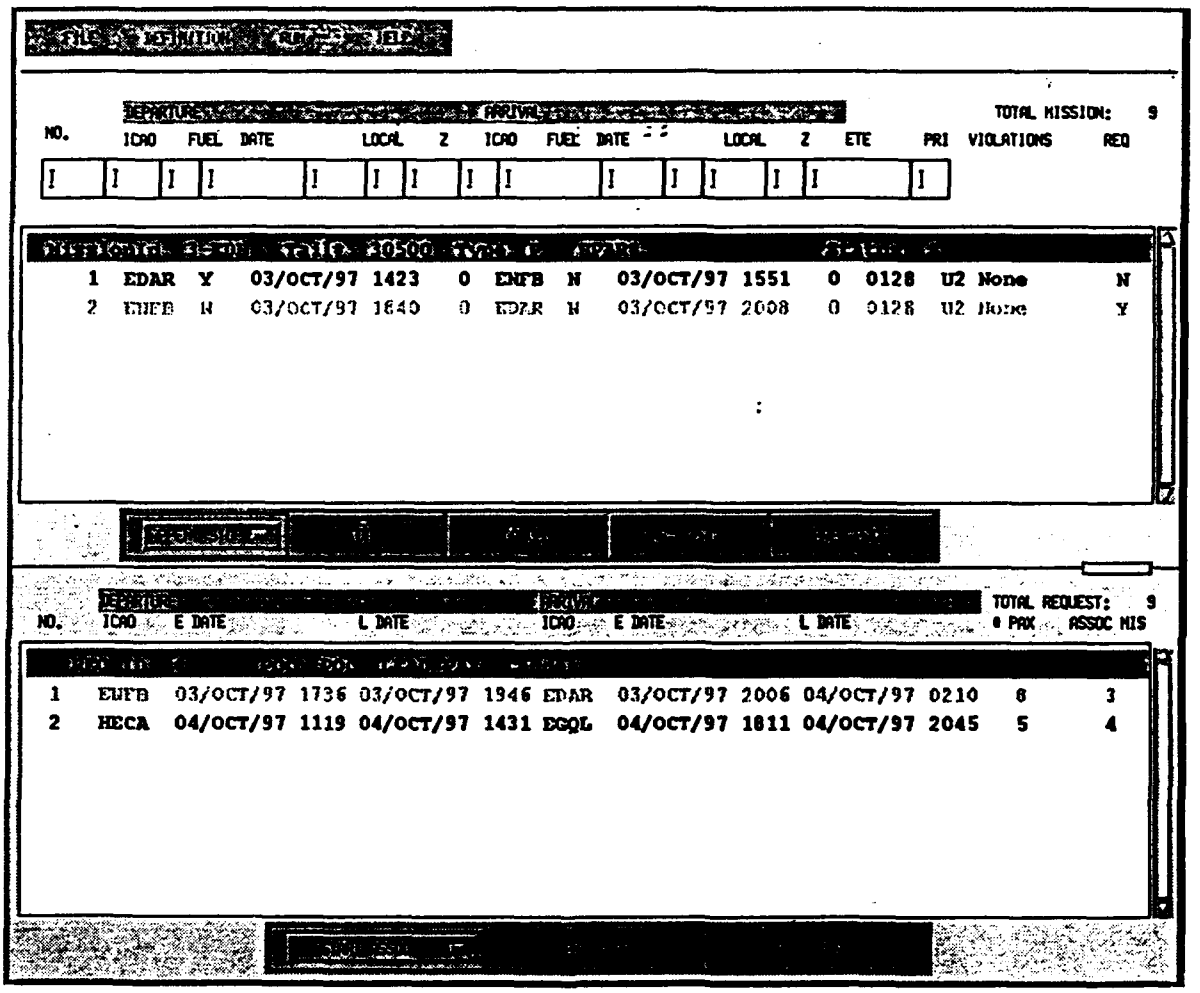

Figure 4.3.2. Clicking on Request 1 highlights Leg 2. This indicates that request 2 is scheduled with Leg 2 of Mission 3.

To unschedule this request click on the highlighted leg in the Mission window. This results in the disappearance of the highlight as shown in *figure 4.3.3*

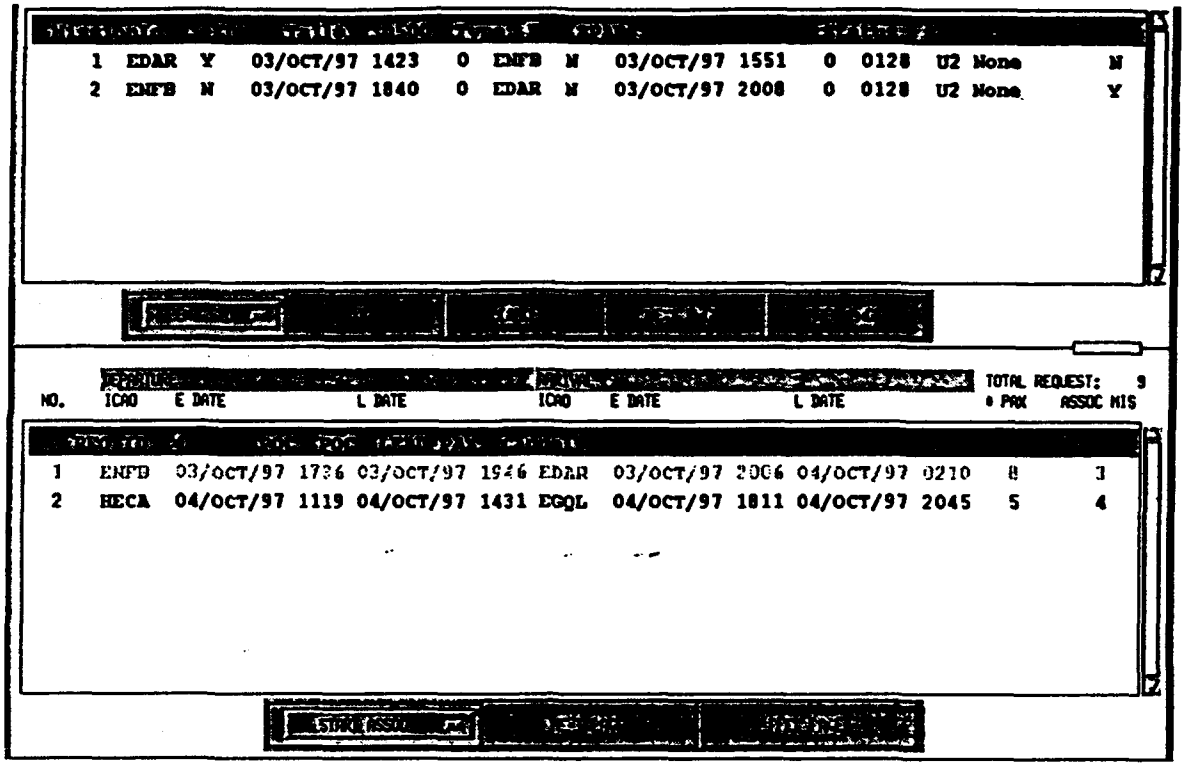

Figure 4.3.3. Unscheduling request 1 from Leg 2 by clicking on Leg 2. The highlighted form disappears.

At this point the leg stands selected to be unscheduled. To confirm the change click on the **START ASSOC** Button again. This pops up a menu as shown in *figure 4.3.4.*

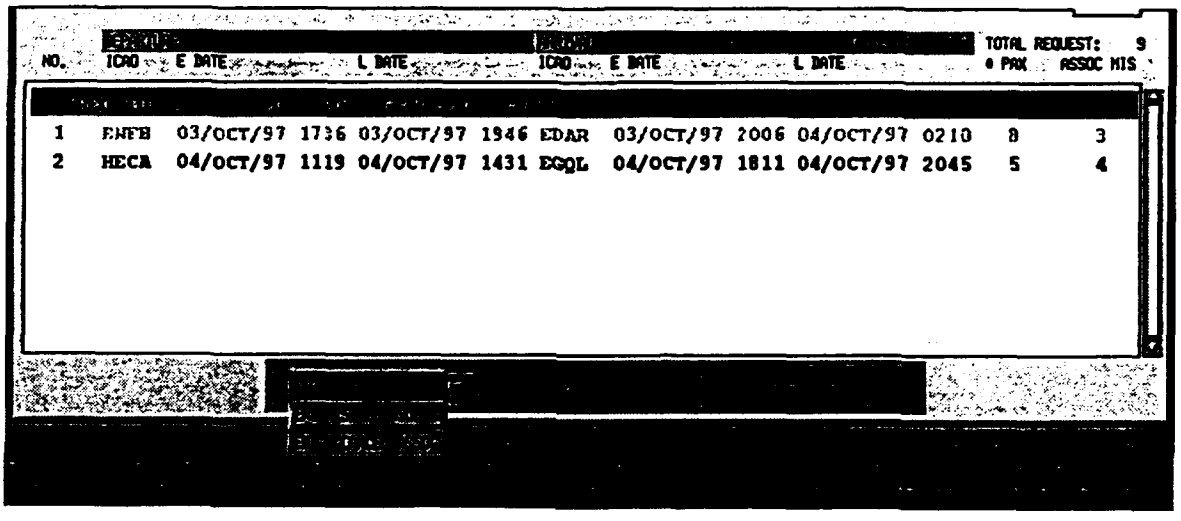

Figure 4.3.4. The pop up menu to SAVE or CANCEL the association.

Now click on **END & SAVE ASSOC.** This saves all the changes and ends the process of unscheduling. The entries of the selected rows change and *NO* leg or request remains highlighted. *Figure 4.3.5* reflects the result of the unscheduling process just done. The numerical value in the **ASSOC MIS** column of the first request *changes to 0*, indicating that this request is no longer associated with a mission. Note that the letter in the **REQ** column of the second leg changes to **N** indicating he leg has no longer a scheduled request.

To cancel the association click on the **END & CANCEL ASSOC.** This undoes the selection and leaves no row highlighted.

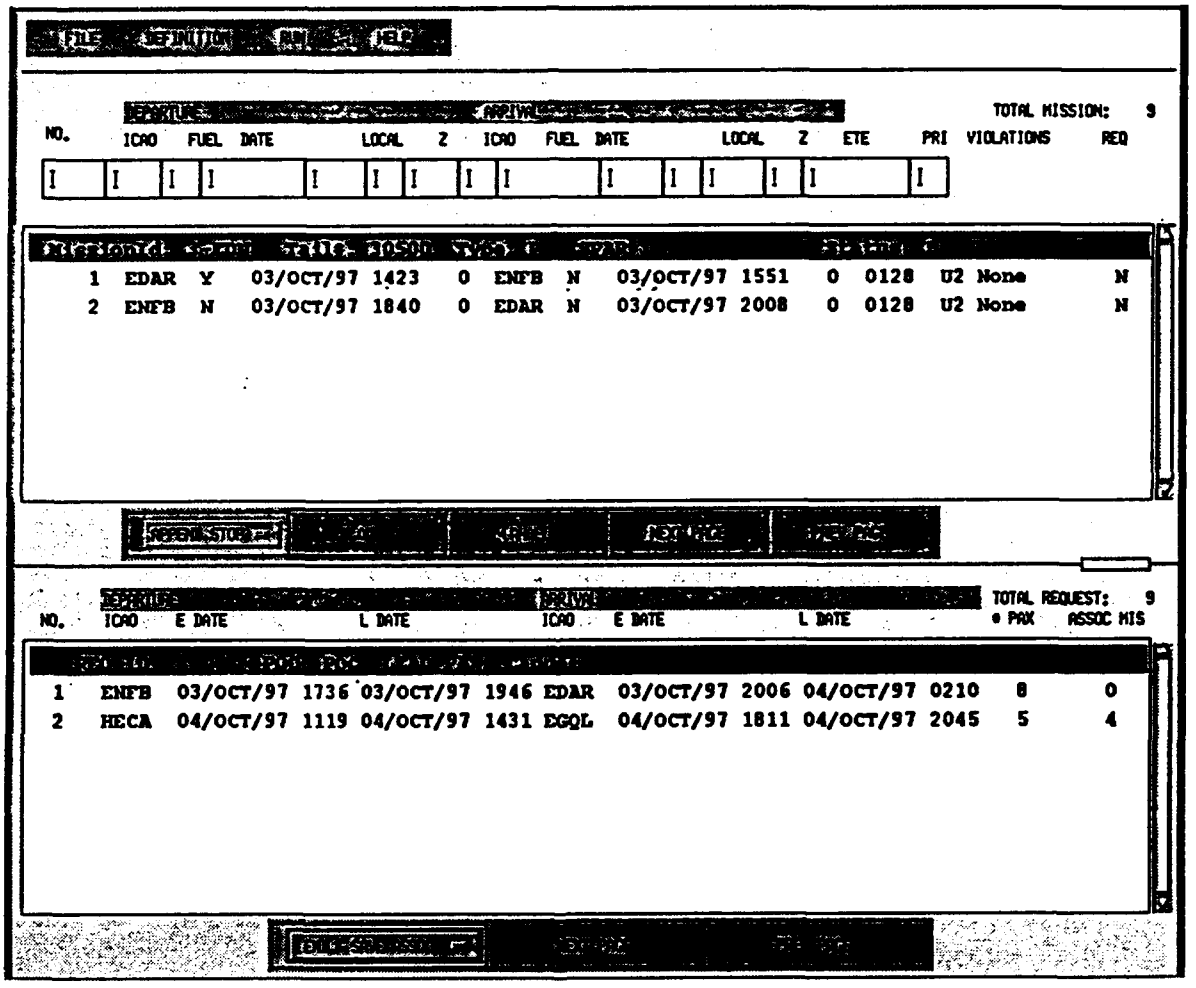

Figure 4.3.5. The result of saving the changes. This completes the process of unscheduling. The change is reflected in the ASSOC MIS column with a 0 entry.

4.3.2 SCHEDULING A REQUEST

It is necessary to have a unscheduled request and a mission for scheduling a request. Consider *figure 4.3.6* which shows the requests with REQ ID 6. The entry 0 in the ASSOC MIS column indicates that the request is not associated with any mission.

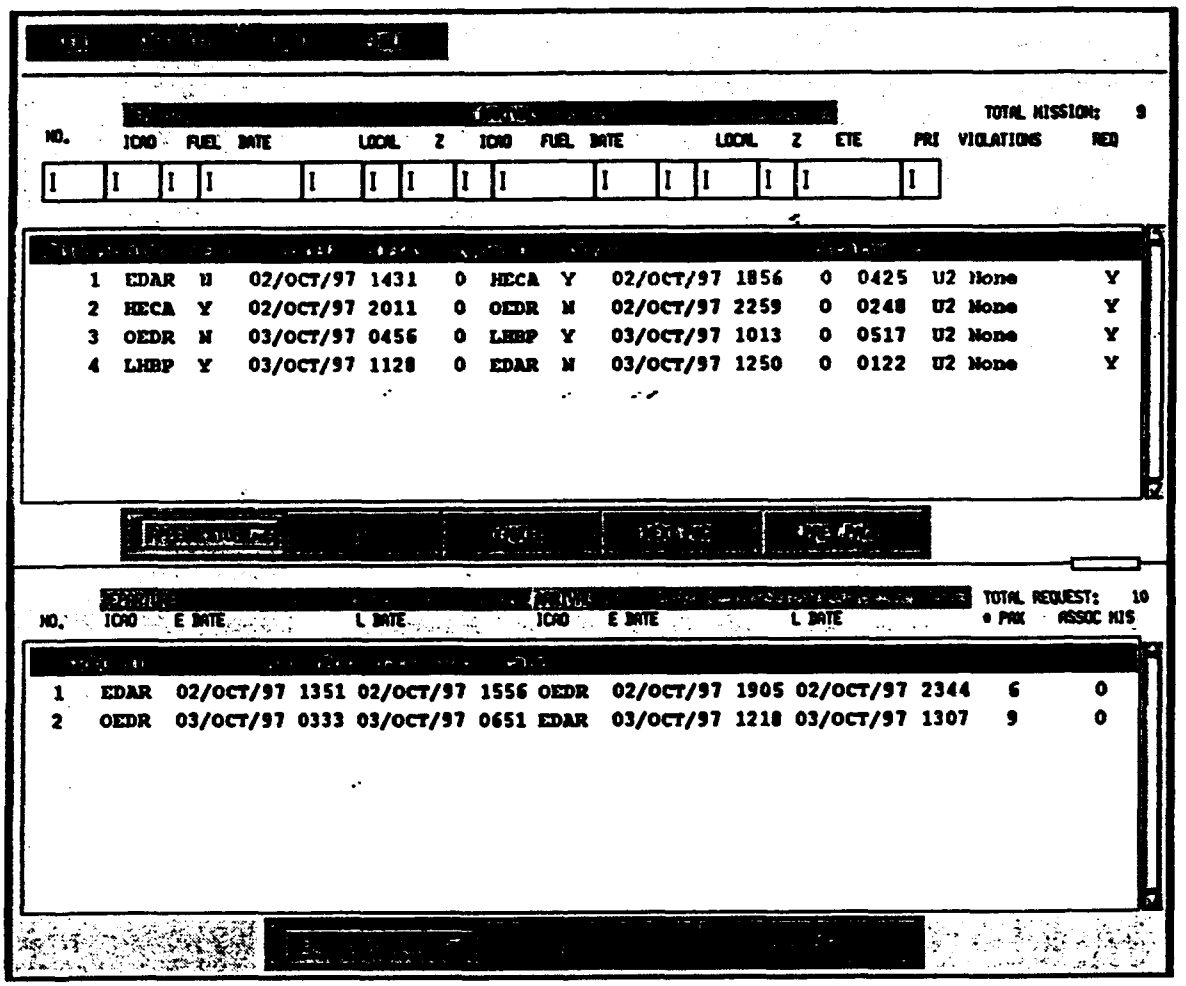

Figure 4.3.6. Initial status of the request 1. 0 in the ASSOC MIS indicates that this request is currently unscheduled.

To start the scheduling click on the leftmost bottom at the bottom and select **START ASSOC**. Refer to the *figure 4.3.4* for the shape of the pop up menu to start association. Click on first row to start association for first request. This row gets highlighted. Based on the *DEPARTURE ICAO* and *ARRIVAL ICAO* Legs 1 and 2 of Mission 5 are selected. To select the legs simply click on them with the left mouse button. The legs get highlighted. This is shown in *figure 4.3.7.*

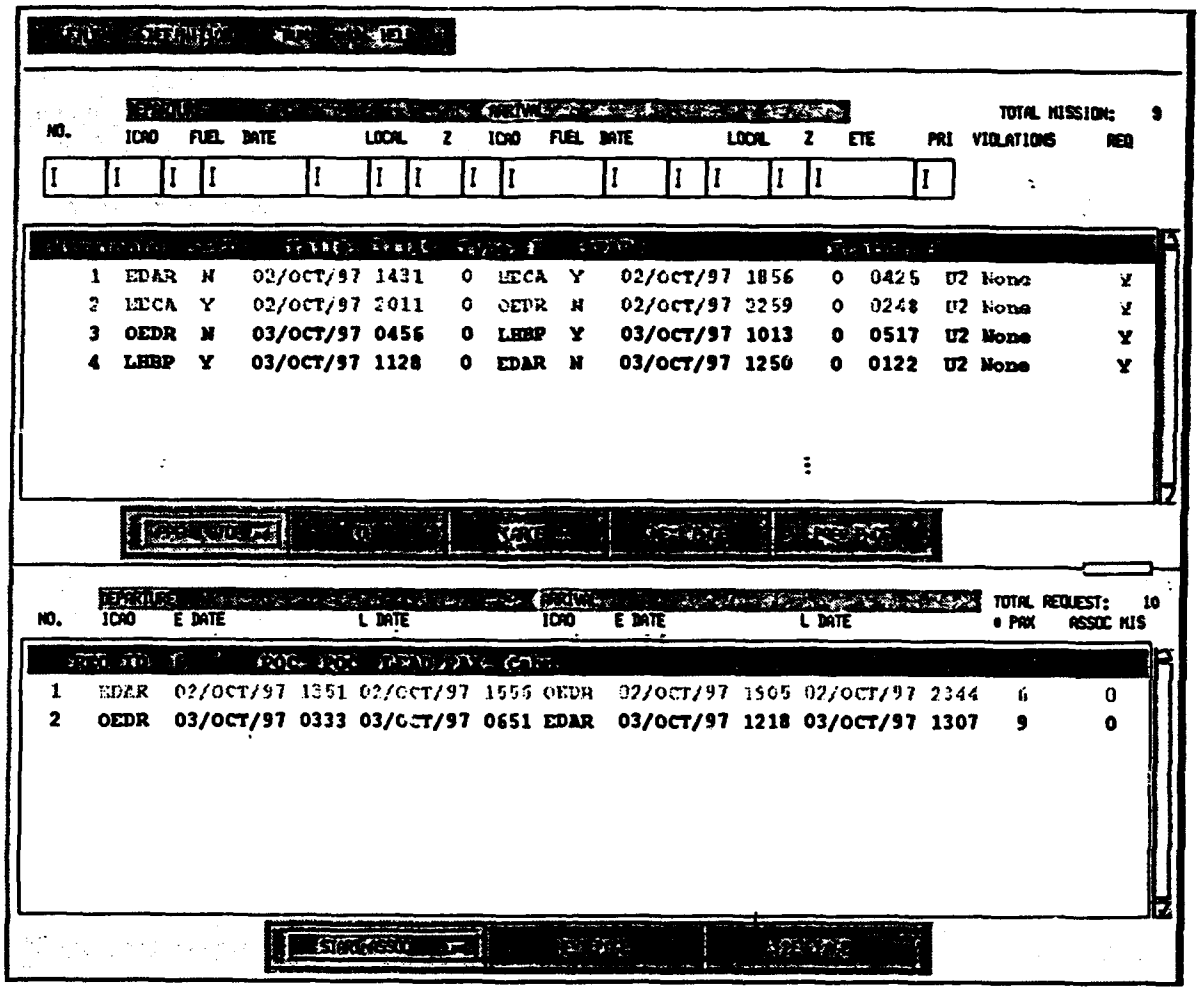

Figure 4.3.7. Leg 1 and 2 get highlighted as a result of clicking on them. This is done to associate Request 1 with Legs 1 and 2 of Mission 5.

To end the association click on the leftmost button at the bottom of the screen. Select **END & SAVE ASSOCIATION** to confirm the scheduling. The entry in the **ASSOC MIS** column gets changed to the mission number of the associated legs. In the example shown in *figure 4.3.8* the column 0 is replaced with 5 to indicate that the request is now associated with Mission 5.

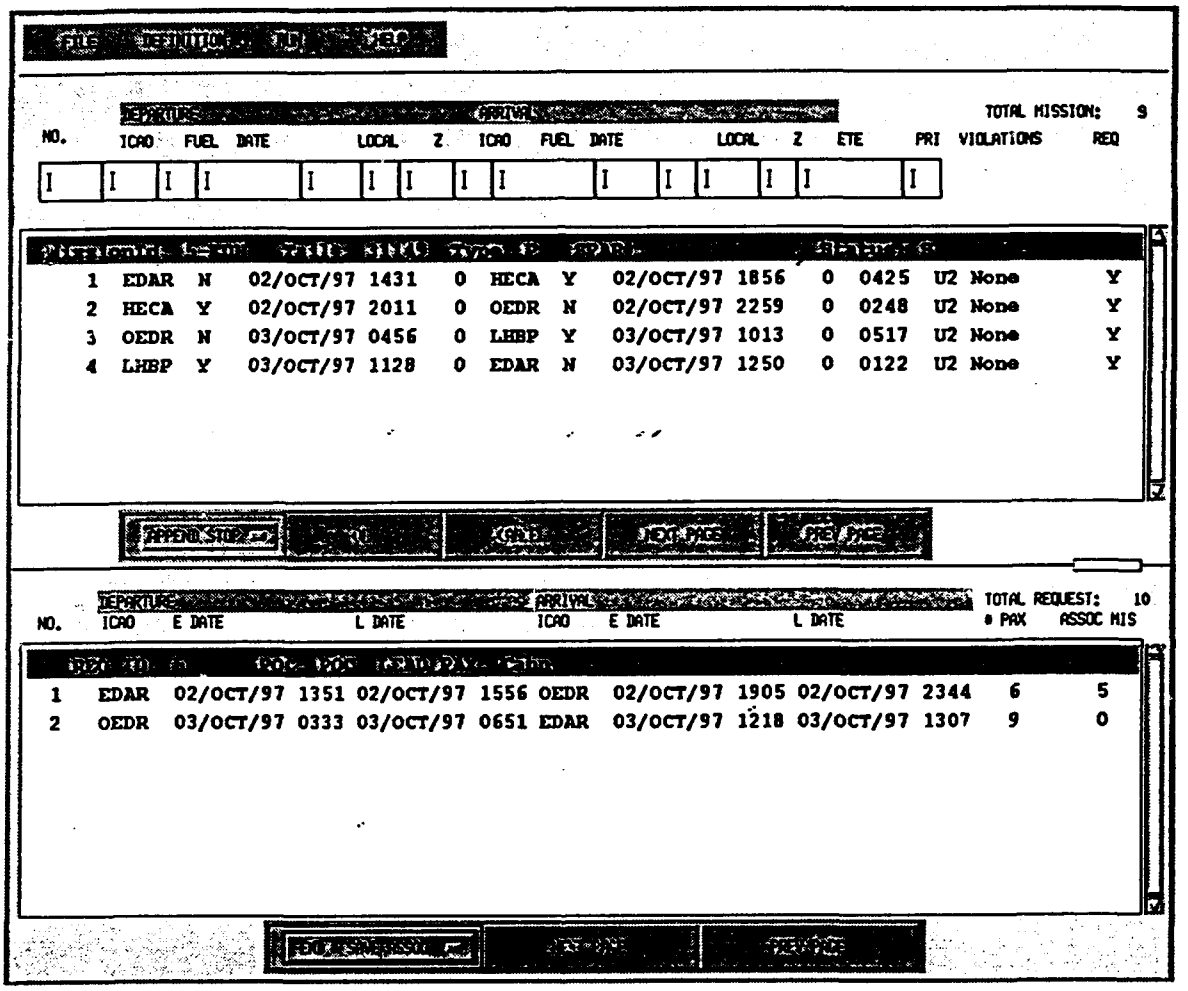

Figure 4.3.8. The completi of Scheduling the request by clicking on END & SAVE ASSOC. The 5 in the ASSOC MIS indicates that the request now is associated with Mission 5.

To cancel the scheduling **END & CANCEL ASSOC** is selected at anytime during the process. This undoes the changes.

The changes can be made permanent to the Database by selecting the option **SAVE SESSION TO DB** from the **FILE** option.

V. INPUT

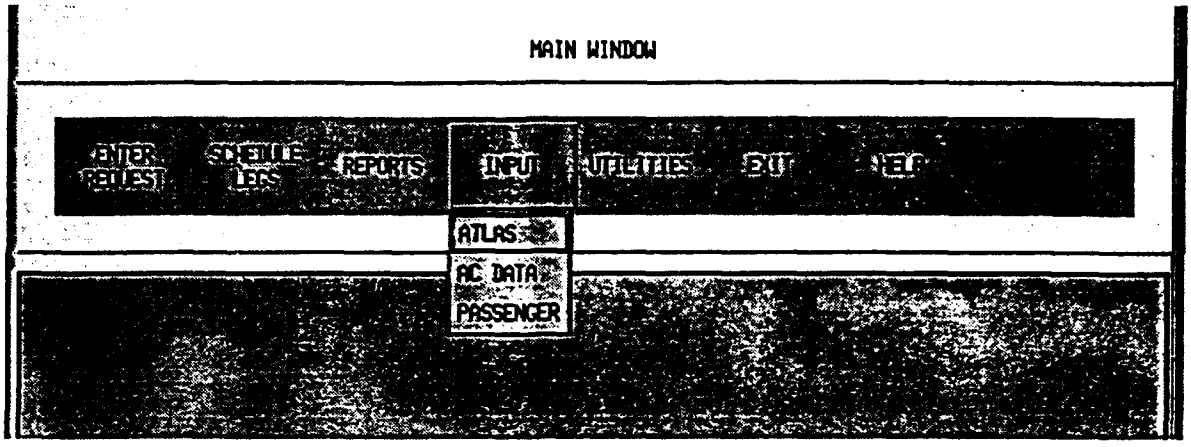

Figure 5. The pull down menu for INPUT from the main window.

5.1 Airport Atlas Screen

Initially, before the software can be fully utilized, data about airports throughout the world should be entered into the database. This is done by selecting the **INPUT** option from the Main Window with the left mouse button. A pull-down menu appears with options to input data for the airport atlas, passenger system or planes. The airport atlas option is selected using the right mouse button. The resultant screen is shown in *Figure 5.1.1.*

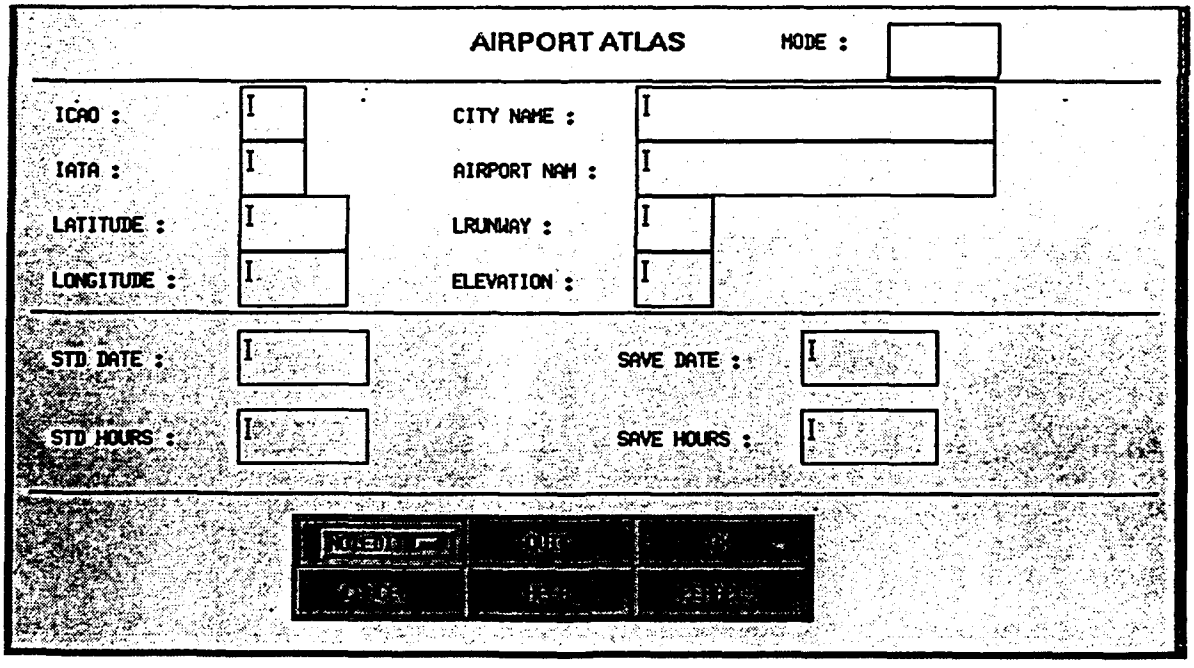

Figure 5.1.1. The window to query on the Airport Atlas Data.

At the bottom of the screen, a button labeled **NO EDIT** appears. Clicking the left mouse button on the NO EDIT results in the display of additional options (Add, Modify, and Delete). *Figure 5.1.2* shows a pop up of this button. The Add option should be selected if information for a new airport is to be added to the system. As a result, the MODE indicator in the upper right corner of the screen changes to ADD. The following information can be added for each airport:

- ICAO, IATA
- City Name
- Latitude, Longitude
- Length of the Runway (LRUNWAY)
- Elevation
- Information about conversion to daylight savings time:
    - Date to convert to standard time (STD DATE)
    - Date to convert to daylight savings time (SAVE DATE)
    - Number of hours to move for standard time (STD HOURS)
    - Number of hours to move for daylight savings time (SAVE HOURS)

After the information has been added, the **OK** button at the bottom of the screen should be selected to save the information to the database. If the user wants to abandon the information without saving, the **CANCEL** button is selected.

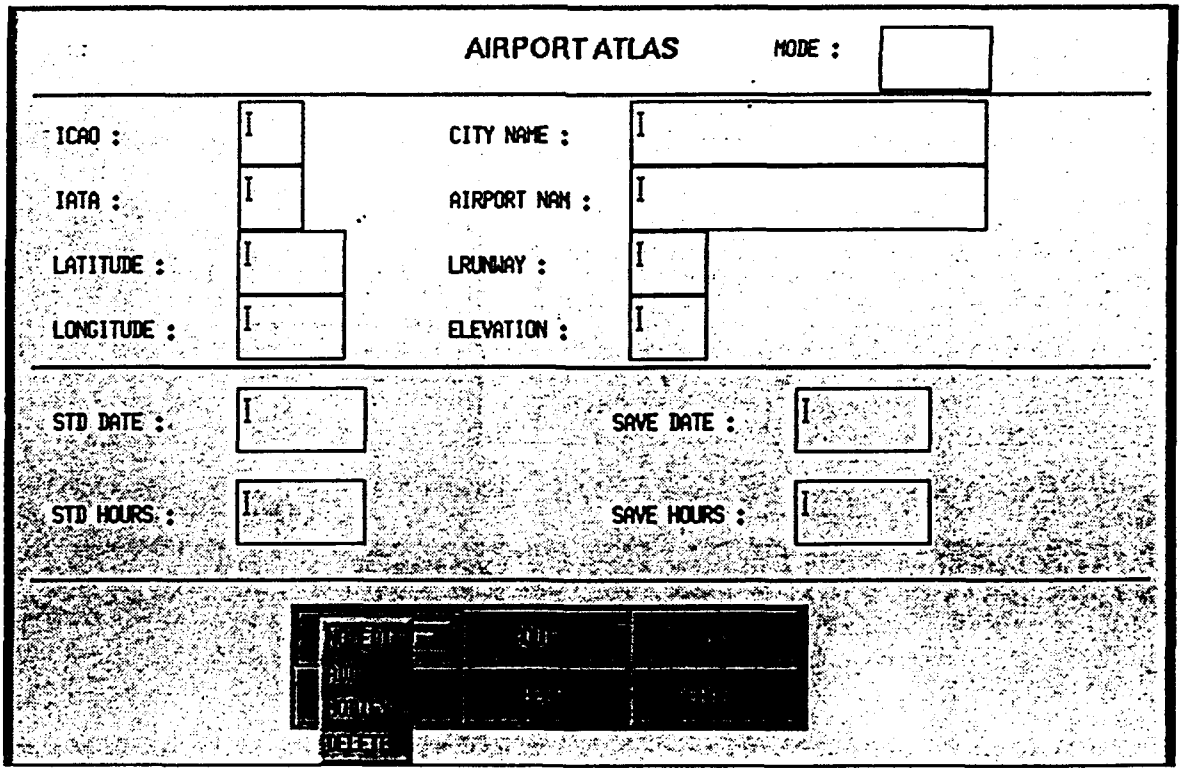

Figure 5.1.2. The appearance of the Airport Atlas screen when NO EDIT button is clicked. The mode(add, modify or delete) is reflected in the box labeled MODE.

Airport information can also be modified or deleted from the system using this screen as well. The **NO EDIT** button is selected, resulting in the display of the options to Modify or Delete. After the user has selected the **MODIFY** option, the **MODE** button at the upper right of the screen reflects the user's choice. The ICAO for the airport and/or the City Name should be entered, each followed by the return key. The other information about the airport will be displayed if the airport exists in the current database, and can be modified as needed. After the appropriate modifications have been made, the **OK** button at the bottom of the screen is selected to save the revised information, or the **CANCEL** button can be used to abandon the changes.

5.2 Aircraft Screen

Data about aircraft used by the system should be entered into the database before it is used. This is done by selecting the **INPUT** option from the Main Window with the left mouse button. A pull-down menu appears with options to input data for the airport atlas, passenger system or planes. The aircraft option (AC Data) is selected using the right mouse button. The resultant screen is shown in *Figure 5.2.1.*

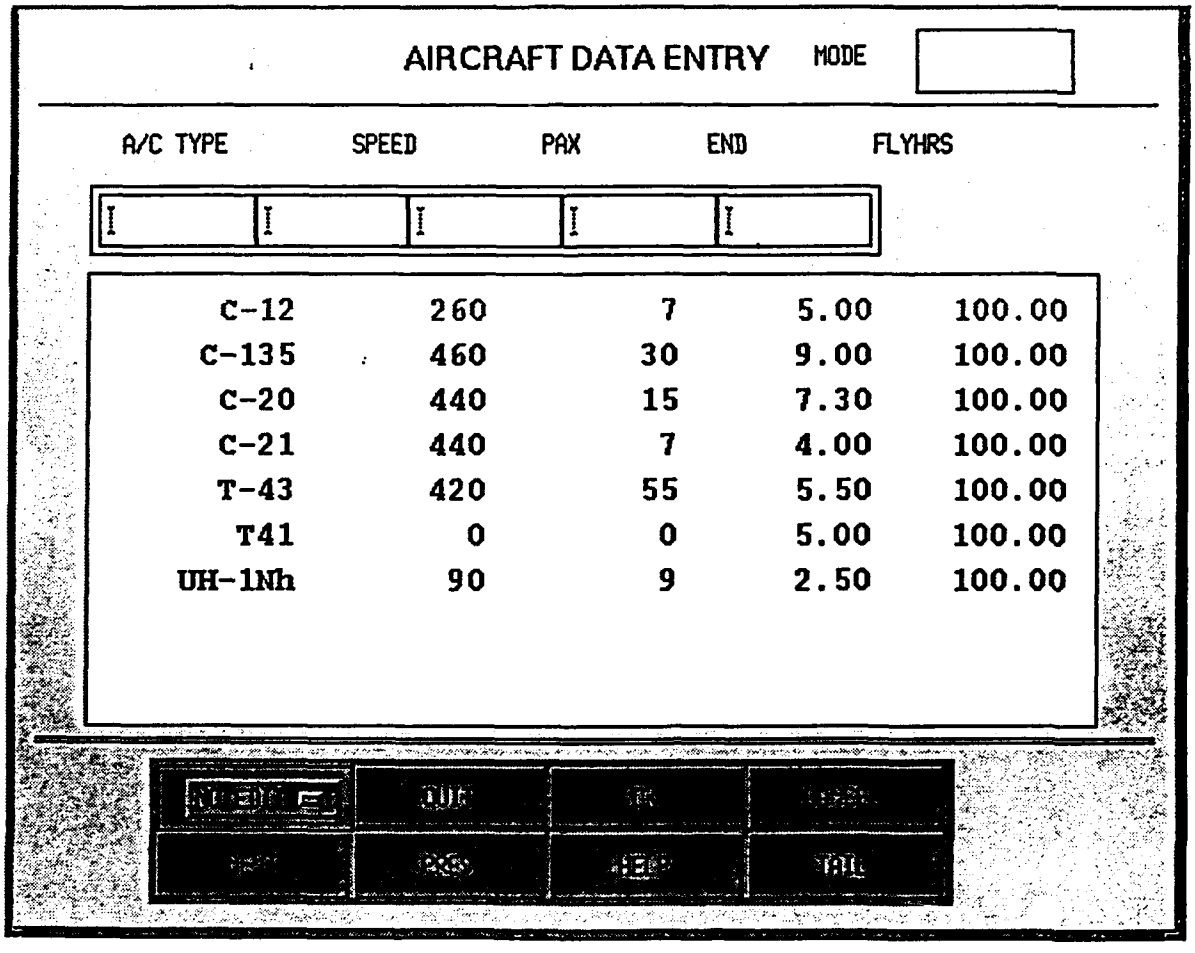

Figure 5.2.1. The Screen for entering Aircraft Data.

At the bottom of the screen, a button labeled **NO EDIT** appears. Clicking the left mouse button on the NO EDIT results in the display of additional options (Add, Modify, and Delete). The Add option should be selected if information for a new plane is to be added to the system. As a result, the MODE indicator in the upper right corner of the screen changes to ADD as shown in *figure 5.2.2*. The following information can be added for each plane type:

- Type of Aircraft (AC/TYPE)
- Speed
- Maximum Number of Passengers (PAX)
- Endurance (END)
- Flying Hours Allotted (FLY HRS)

After the information has been added, the **OK** button at the bottom of the screen should be selected to save the information to the database. If the user wants to abandon the information without saving, the **CANCEL** button is selected. The **NEXT** and **PREV** buttons allow the user to page through all available aircraft types already in the system.

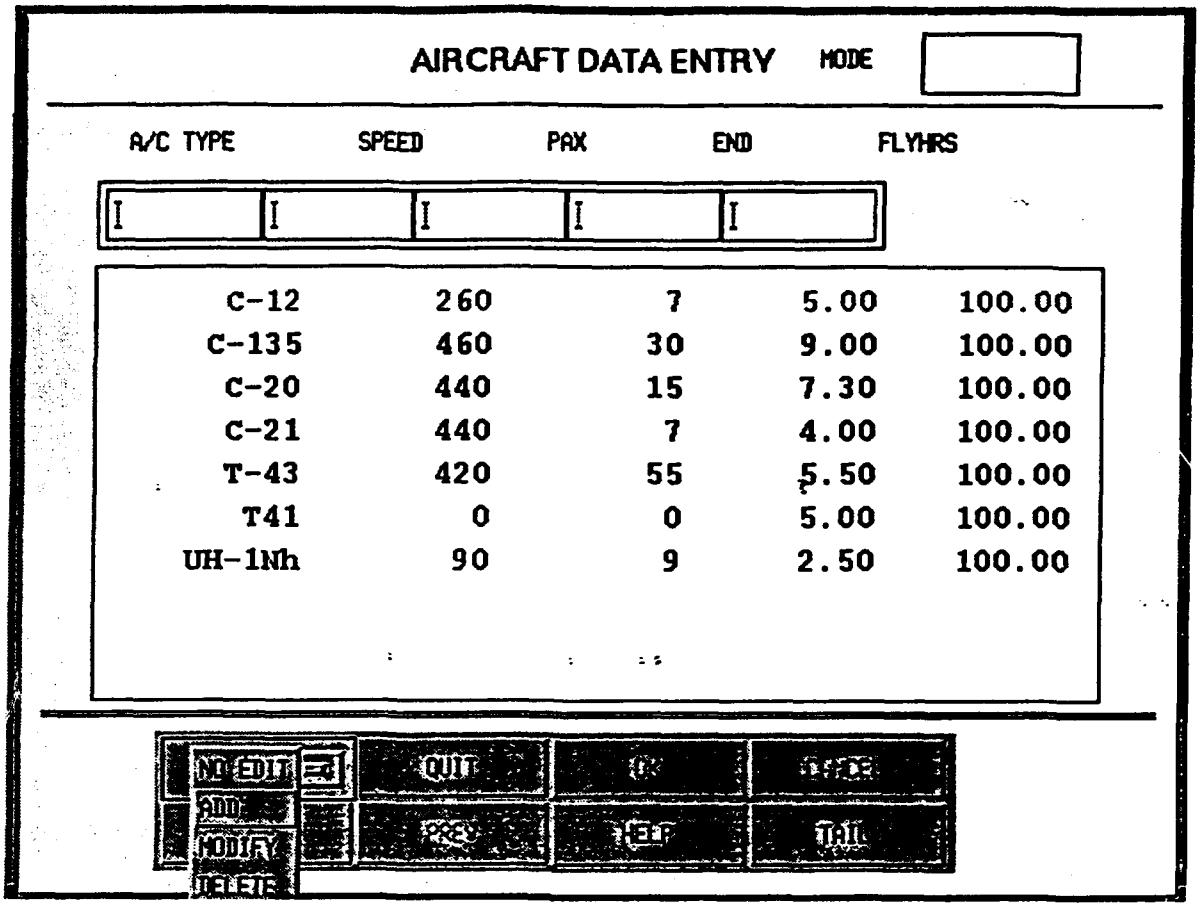

Figure 5.2.2. Aircraft Data Entry Screen during the entry of a data. The mode (add, delete or modify) is reflected in the box labeled MODE. The fields under the labels A/C TYPE, SPEED, PAX, END and FLYHRS are used for adding and modifying the data. The data to be deleted gets reflected here.

The TAIL button allows specific tail numbers to be entered for each plane type. Selecting the TAIL button produces the screen shown in *Figure 5.2.3*, which allows tail numbers to be added, modified, or deleted. Choosing the ADD option allows the user to add a tail number for a particular plane type. After the information about tail number and plane type have been entered, the OK button is used to save the information, or the CANCEL button can be used to abandon without saving. Tail numbers can also be modified or deleted using this screen.

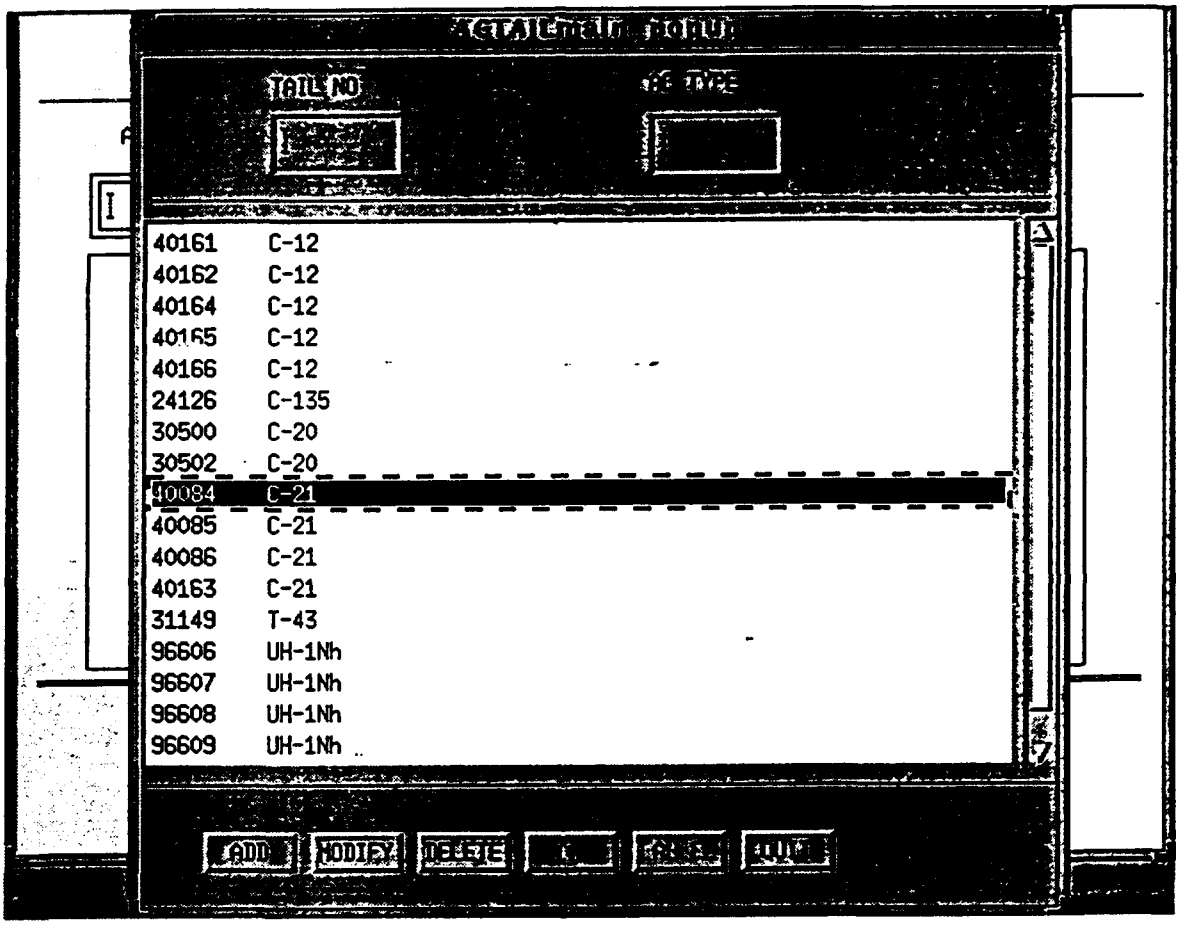

Figure 5.2.3. This figure shows windows pulled out on clicking on the TAIL button of the window shown in figure 5.2.2. This screen is used for adding a tail number for a particular plane type.

Aircraft information can also be modified or deleted from the system using this screen as well. The **NO EDIT** button is selected, resulting in the display of the options to Modify or Delete. To modify information, select the plane type to be modified using the left mouse button *BEFORE* choosing the MODIFY option from the bottom row of buttons. After the user has selected the MODIFY option, the *MODE* button at the upper right of the screen shows MODIFY and the information for that plane is retrieved into the boxes along the top of the screen, if that plane type exists in the current database, and can be modified by the user as needed. After the appropriate modifications have been made, the **OK** button at the

bottom of the screen is selected to save the revised information, or the CANCEL button can be used to abandon the changes.
The process is repeated to delete a plane type from the database. First, the plane type to be deleted is selected, and then the DELETE option is chosen from the buttons along the bottom of the screen. The information about that plane type is displayed in the boxes along the top of the screen, and if the user decides to actually delete that plane type, the OK option is selected. Otherwise, CANCEL is used to abandon the operation.

The QUIT option is used to exit from this screen.

5.3 Passenger Screen

Data about passengers who frequently use the system can be entered into the database if desired. This is done by selecting the INPUT option from the Main Window with the left mouse button. A pull-down menu appears with options to input data for the airport atlas, passenger system or planes. The passenger option (PAX) is selected using the right mouse button. The resultant screen is shown in *Figure 5.3.1.*

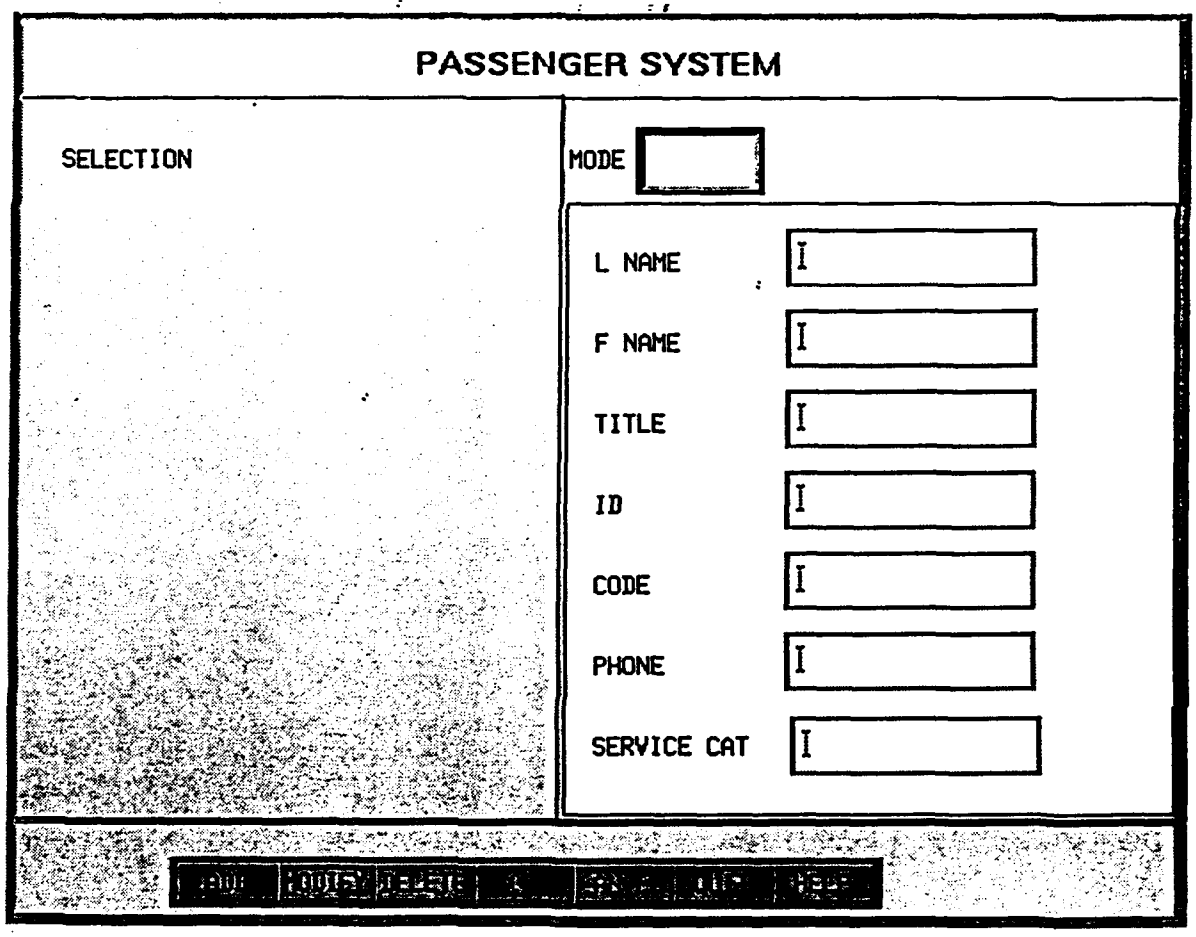

Figure 5.3.1. The Passenger System screen. The buttons at the bottom of the screen can be used to add, modify or delete a record.

The Add option should be selected if information for a new passenger is to be added to the system. As a result, the MODE indicator in the upper right corner of the screen changes to ADD. The following information can be added for passengers:

- Last Name (L Name)
- First Name (F Name)
- Title
- Identification Number (ID)
- Code
- Phone
- Service Category (Service Cat)

After the information has been added, the **OK** button at the bottom of the screen should be selected to save the information to the database. If the user wants to abandon the information without saving, the **CANCEL** button is selected.

After the user has selected the MODIFY option, the MODE button at the upper right of the screen reflects the user's choice. The Last Name (L NAME) and the First Name (F NAME) should be entered, each followed by the return key. Simply pressing the <enter> key for both the last and first name fields retrieves all passengers in the database. If only the first letter of the last name is known, that can be entered followed by a <enter> and in the first name field, a <return> can be entered. This will retrieve all passengers whose last name begins with the letter typed. An example is shown in *figure 5.3.2*. After the appropriate passenger name is selected using a double-click of the left mouse button, the other information about the passenger will be displayed in the boxes along the right side of the screen, and can be modified as needed. After the appropriate modifications have been made, the **OK** button at the bottom of the screen is selected to save the revised information, or the **CANCEL** button can be used to abandon the changes.

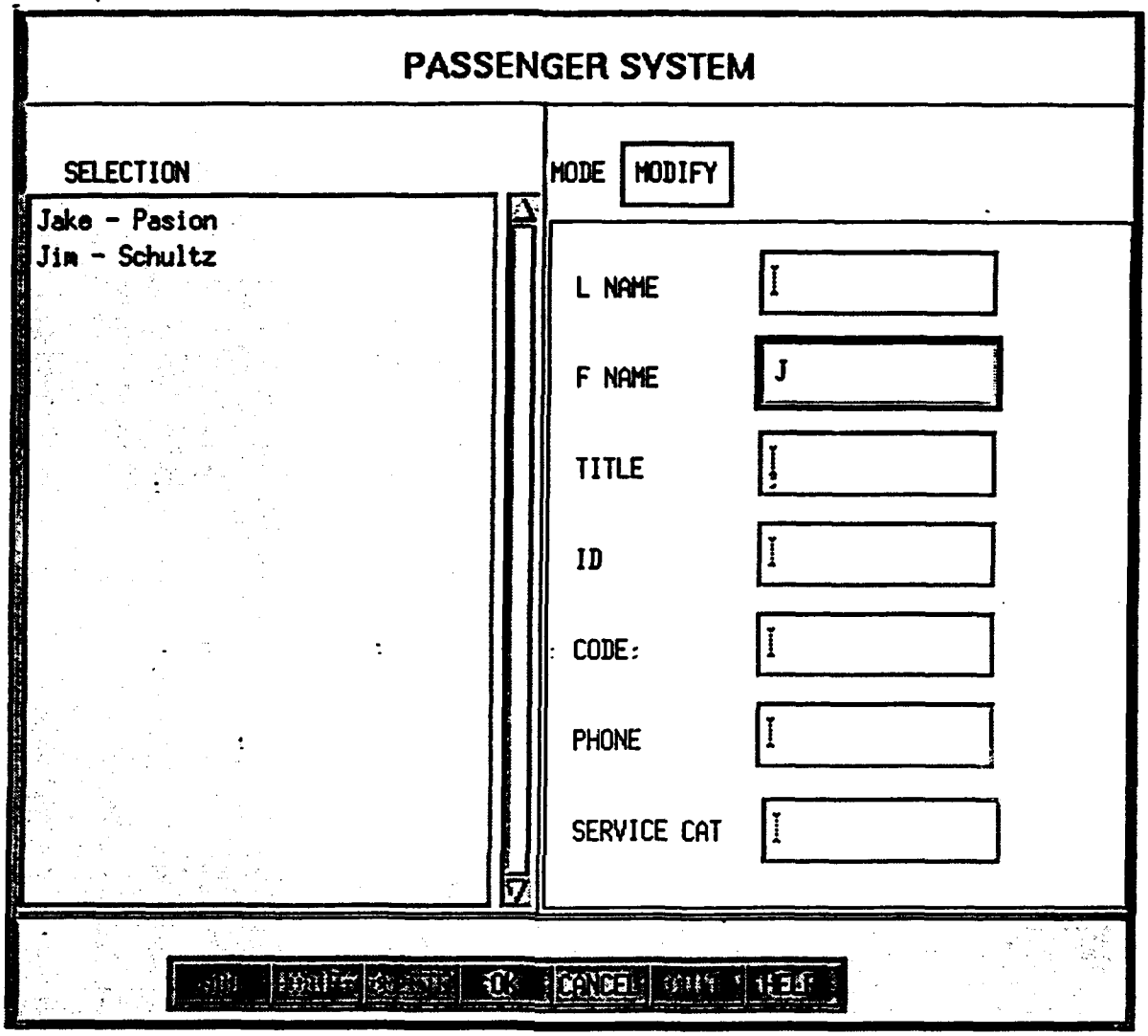

Figure 5.3.2. An example of the information retrieval from the Passenger Screen

The QUIT option is used to leave this screen.

VI. DATABASE SUPPORT IN DAKOTA

6.1 Database Schema

Figure 6 presents the schema diagram of the database used for DAKOTA. Following the entity relationship approach fourteen tables are maintained. The schema file is created based on this table using the Database Management System, presented in section 3. Table Aircraft_type stores the details such as type, speed, capacity, endurance and flying hours. Airport_atlas as the name implies stores such details as city_name, airport_name, latitude, longitude, etc. The data manipulation on this table can be done using the Aircraft Data Screen explained in section 4.2. The other table of interest is passenger. This stores the details of each passenger with reference to a unique identification and details such as last name, first name, service category, dv_code, phone and title. While working with scheduling missions a number of tables are used by the database depending on the type of query. Tables that store information about the missions are Mission, Mission_legs and Mission_revision. Passenger and Pax_group store information about the Personnel. Priority describes the priority for each code. Tables that maintain information about various requests are Request, Request_legs, Request_mission and Request_revision. Table Scheduler stores information about the name of the scheduling agent each with a unique identification. Table Tail maintains information regarding tail of an associated mission and legs.

Aircraft_type

ac_type	speed	capacity	endurance	fly_hours

Airport_atlas

icao	iata	city_id	city_name	airport_name	latitude	longitude

elevation	max_runway	std_date	std_hours	sav_hours

Mission

id	char_id	opid	enter_date	last_mod	tail_num	status

start_date	end_date	spar_num	mission_type	actual_hours

Mission_legs

mission_id	leg_id	flying_date	orig_icao	dest_icao	dept_time

arrv_time	orig_fuel	dest_fuel

Mission_revision

mission_id	remark	rev_dt	scheduler

Passenger

id	usafe_id	title	fname	lname	service_cat	dv_code	phone

Pax_group

request_leg_id	pax_id

Priority

code	prority

Request

request_id	ac_type	poc	wphone	hphone	paxid	req_date

req_agency	priority_code	remarks1	remarks2	remarks3

Request_legs

request_id	leg_id	num_pax	orig_icao	dest_icao	e_dept_time	l_dept_time

e_arrv_time	l_arrv_time	e_dept_stat	l_dept_stat	e_arrv_stat	l_arrv_stat

specification	release_seats

Request_mission

request_leg_id	mission_leg_id

Request_revision

request_id	remark	rev_dt	scheduler_id

Scheduler

id	name

Tail

tail_num	ac_type

Figure 6 Database Schema Tables. The figure describes the relational schema of the database used for DAKOTA.

6.2 Database Query Functions

This section discusses the database functions used by the system for information retrieval. While Query functions retrieve information without affecting the database, functions such as tail_add, tail_delete, etc., modify the database. Section 6.2.1 discusses the query functions and section 6.2.2 lists out all the database manipulation functions used by the Scheduling system.

6.2.1 QUERY FUNCTIONS TO OPERATE ON USAFE DATABASE

Aircraft_type_query : This query collects details of aircrafts from the table aircraft_type.
ex: aircraft_type_query("C", num_records, record_struct) will collect all the rows where the aircraft_name starts with "C".

Calc_flying_hours : This finction claculates the time of flying hours of aircrafts in a particular mission leg and tail leg. The information about aircraft-type Month year are collected.
ex : calc_flying_hours ("10/10/93", Buffer) will collect information from the tables aircraft_type, mission, tail, mission_legs where year of dept_time (table mission_legs) = 1993.

Get_mission_bounds : Function to collect bounds on mission that involves tail_number, start_time, last_mission_icao, next_mission_icao, previous and next latitudes and longitudes of missions, last and next mission_id. This function aids in calculating the ET and LT times for mission headers.
ex :get_misssion_bounds(1234, &mission_start_time, &ET, &LT, last_mission_icao, next_mission_icao, &originate, &terminatate, &prev_miss_id, &next_miss_id, &next_miss_id) will collect information about latitude, longitude from table airport_atlas and mission_id, orig_icao, dest_icao, dept_time, arrv_time from table mission_legs and the following conditions will be satisfied :tail_num (table mission) = 1234, id (table mission) = mission_id (table mission_legs), orig_icao (table mission_legs) = icao (table airport_atlas) and dest_icao (table mission_legs) = icao (table airport_atlas).

Get_first_mission_leg : Function to retrieve a record from the table mission_legs for the required mission_id. The details are regarding flying_date, orig_icao, destination_icao, departure_time and arrival_time.
ex : get_first_mission_leg(9999, &miss_leg_record) will collect a record where mission_id (table mission_legs) = 9999.

Get_mission_tail_num : Function to retrieve the tail_number from table mission and departue_time from table mission_legs.
ex : get_mission_tail_num(9999, &tail_num, date_string) will assign tail_num value of tail_num (table mission) and date_string value of dept_time (table mission_legs) whereid (table mission) = 9999 and id (table mission) = mission_id (table mission_legs)

Grease : This Function displays information about scheduled missions within a range of dates for display in a "greaseboard" window.The Input: 2 dates in the form dd/mmm/yy.The output:is a buffer of info to display the greaseboard
ex : grease("10/JAN/93", "15/FEB/93", Buffer) will collect information about all the missions that are scheduled between January 15, 1993 and February 15, 1993.

MapByTail : Function to collect information about legs on a given mission for "View By Tail/Spar #" (upper right window). The Input: a mission number and Output is a Buffer containing info about legs/Pax (bottom left text window)
ex : mapByTail(9999, Buffer) will collect information about legs having the mission-id = 9999 in table mission_legs.

MapmAct : Function to collect info about planes undergoing maintenance on "myDate" for display in default "schedule request" window.Input: a date in the form dd-mmm-yy, a plane type (character string) an aircraft type (character string) a pointer to a buffer to be filled with information. Output: A buffer containing info needed to display information about planes in maintenance on the given day.
ex : mapmAct ("15-OCT-93", "C-21", BUFFER) will collect information from the table tail and mission where mission_type (table mission) = "M",start_date (table mission) October 15, 1993 and ac_type (table tail) = "C-21"

MappAct : Function to collect info for display for the "View by plane type" .Input: an integer indicating the view (PLANE-VIEW or SPAR_VIEW),a date in the form dd-mmm-yy (char string), and an aircraft type (character string). an address of the pointer that will point to the returned structures an address of an integer which will hold the number of missions returned Output: A structure (rtInfo) containing info needed to plot scheduled missions on the map and the number of scheduled missions for the given day.There will be "totMissions" structs of type RTINFO returned in an array pointed to by "*rtInfo".
ex :mappAct(SPAR_VIEW, "06-OCT-93", "353", &RTINFO, &num_records) will collect information from tables mission, mission_legs, tail, reuest, request_legs, airport_atlas where id (table mission) = 353, id (table mission) = mission_id (table mission_legs), tail_num (table tail) = tail_num (table tail), leg_id (table mission_legs) = mission_leg_id (table request_mission), leg-id (table mission_legs) = request_leg_id (table request_mission), orig_icao (table mission_legs) = icao (table airport_atlas), dest_icao (table mission_legs

) = ico (table airport_atlas), id (table mission) = mission_id (table mission_legs), mission_type (table mission) = mission-id (table mission_legs) and mission_type (table mission) = "P"

MaprAct : Function to collect info about reserved planes on "myDate" for display in default "schedule request" window. Input: a date in the form dd-mmm-yy, a plane type (character string) an aircraft type (character string) a pointer to a buffer to be filled with information Output: A buffer containing info needed to display information about reserved planes on the given day.
Ex: maprAct("06-OCT-93", "C-20", Buffer) will collect information from table tail, mission where ac_type (table tail) = "C-20", start_date (table mission) <= October 6, 93, end_date (table mission) >= October 6, 93 , tail_num (table tail) = tail_num (table tail_num), mission_type (table mission) = "R". The records are odered by ac_type and start date.

MapuAct(char *myDate,char *myAcType,char *buf) : Function to collect info about unassigned planes for a given date for display in "View by plane type window" Input: a date in the form dd-mmm-yy, a plane type (character string) an aircraft type (character string) a pointer to a buffer to be filled with information Output: A buffer containing info needed to display information about unassigned planes
ex : mapuAct ("06-OCT-93", "C-21", Buffer) wil collect information about all the unassigned planes for October 6, 93.

MapmAll : This Function collects information about planes undergoing MAINTENANCE on "myDate" for display in default "schedule request" window.The input: is a date in the form dd-mmm-yy.and the output: is a buffer containing info needed to display information about planes in maintenance on the given day.
ex: mapmAll("18-JAN-93", BUF) will provide the information of all the planes in maintenanace on the day January 18,1993.

MappAll : This function collects information about *SCHEDULED MISSIONS* on "myDate" for display in "schedule request" window. The input is a date in the form dd-mmm-yy. And the output is a structure (rtInfo) containing info needed to plot scheduled missions on the map and the number of scheduled missions for a given day.
ex :mappAll("16-OCT-1993", &rtInfo, totMissions) will collect details of all the missions on October 16,1993.

MaprAll : This Function collects information about *RESERVED* planes on "myDate" for display in "schedule request" window. The input is a date in the form ddd-mm-yy.and output is a buffer containing info needed to display information about reserved planes on the given day.
ex : maprAll("10-JAN-93", buffer) will collect information about all the reserved planes on January 10,1993.

MaprqAll : This function to collect information about requests on "myDate" for display in default "schedule request" window (upper right window)The Input is a date in the form dd-mmm-yy. And output is a structure (rqstInfo) containing info needed about requests. Here the latitude and longitude are stored in the database as a double, but they MEAN degrees.minutes. Hence, a conversion is done to convert the lat/long from degrees and minutes into degrees.
ex : maprqAll("10-JAN-93", &StructINFO, &Num_requests) will collect information about all the requests on January 10, 1993.The no of requests is given by Num_requests.

MapuAll : This function collects informatio about UNASSIGNED planes for a given date for display in "schedule request" window. The input is a date in the form dd-mmm-yy. The output is a buffer containing info needed to display information about unassigned planes
ex: mapuAll("15-JAN-93", Buffer) will collect the information about the planes that are assigned on January 15,1993.

MissionSel : Function to select records from the tables mission and tail based on id, two dates, mission_type and tail_number.
ex : MissionSel(-1, "10/06/1995", "10/16/1995", &miss_list, &num_missions) will collect all the records from the tables mission and tail between October 6, 1995 and October 16, 1993, with mission_type = "p" and tail_num (table tail) = tail_num (table mission).MissionSel(23, "10/06/1995", "10/16/1995",&miss_list, &num_missions) will collect all the records from the table mission and tail where id (table mission) = 23 and tail_num (table tail) = tail_num (table mission).The records are ordered by id.

Passenger_query : Function to collect details of all the passengers with matching lastname or first name or both. If '*' is entered for the name then all the enteries will be selected.
ex : passenger_query("C", "P", &num_records, &Struct) will collect all the passengers whose last name start with "C", and first name starts with "P". The num_records will have a value that is equivalent to the number of records selected. if ("*", "*", &num_records, &Struct) will collect the details of all the passengers.

Passenger_queryid : Function that retrieves passenger records based on a unique id. Similar to passenger_query but the query is done on the id values of the records.The results will be sorted on last name and first name.
ex : passenger_queryid(9999, &num_records, &Struct) will select all the records where the id value is 9999.

Request_legs_query : Function to retrieve all the records based on request_id. from table request_legs.
ex : request_legs_query(9999, &num_records, &total_pax, &req_paxs, &info) will collect all the records with request_id = 9999. total_passengers indicates the total number of passengers. The info variable acts as a flag indicator that will invoke another query when set that will query : request_passenger_query.

Retrieve_mission : Function to retrieve information from the tables mission, tail, aircraft_type for a particular mission_id(table mission_id).
ex :retrieve_mission(9999, &mission_structure) will gather information about id, enter_date, speed, capacity, endurance, mission_type, status, etc. where id (table mission) = 9999 ,tail_num(table mission) = tail_num (table tail) and ac_type (table aircraft_type) = ac_type (table aircraft_type)

Retrieve_mission_legs : Function to retrieve information about mission_legs from tables mission_legs, airport_atlas
ex :retrieve_mission_legs(9999, &num_legs, &mission_leg_structure) will collect all the records where mission_id (table mission_legs) = 9999, orig_icao (table mission_legs) = icao (table airport_atlas) dest_icao (table mission_legs) = icao (table airport_atlas)

Request_passenger_query : Function to retrieve records from table passenger and pax_group based on request_leg_id.
ex : request_passenger_query(9999, &num_records, &Struct, &total_pax, &PasStruct) will collect records from the table.passenger and pax_group where id (table passenger) = paxid (table pax_group) and request_leg_id (table pax_goup) = 9999.

Request_query : Function that retrieves records based on id or point of contact and lead passenger.
ex : request_query(9999, "*", "*", &num_records, &Struct) will collect records of all the passengers with reuest_id = 9999 (table request) and id (table passenger) = paxid(table request).request_query(-1, "P", "C", &num_records, &Struct) will collect records of all the passengers with point-of-contact (table request) matches P, last_name(table passenger) matches C and id (table request) matches paxid (table passenger).

Retrieve_request_legs : Function to retrive information about request_legs from the tables request, request_legs, request_mission, airport_atlas, passenger for a certain leg_id.
ex : retrieve_request_legs(1, 9999, &num_records, &req_leg_structure) will collect all the records where leg_id (table request_legs) = 9999, request_id (table request) = request_id (table request_legs), orig_icao (table request_legs) = icao (table airport_atlas), dest_icao (table request_legs) = icao (table

airport_atlas), paxid (table request) = id (table passenger) and leg_id (table request_legs) = request_leg_id (table request_mission),

Tail_query : Function to query on the table tail. The query will select all the records from the table aircraft_type and table tail based on ac_type.or tail_number.
ex : tail_query(1234, "P", &num_records, &Structure) will collect all the records from table tail and aircraft_type where tail_num (table tail) = 1234 and ac_type (table tail) = ac_type (table aircraft_type). tail_query(-1, "P", &num_records, &Structure) will collect all the records from table tail and aircraft_type where ac_type (table aircraft_type) matches "P" and ac_type (table tail) = ac_type (table aircraft_type)

6.2.2 DATABASE MANIPULATION FUNCTIONS

Aircraft_type_add : To add a new set of records to the table aicraft_type.

Aircraft_type_update : To update the record of a particular aircraft.

Aircraft_type_delete : To delete an aircraft entry from the database.

Mission_add : To add details about a mission to the database. The detail includes mission identification, operation_id, enter_date, last_mod, tail-number, status, start-date, end-date, spar-num, mission_type, and actual-hours etc.

Mission_update : Update the details of a mission

Mission_delete : To delete a record from the mission.

Mission_leg_add : To add a new set of details of a particular mission_leg to the database. The details added include the mission_identification number, leg_id, origin-icao, dest-icao, departure-time, arrival-time, origin-fuel, destination_fuel.

Mission_leg_delete : To delete a record from the mission_legs.

Mission_leg_update : To update details of a particular row in mission_legs.

Pax_group_add : Function to add an entry to the table pax_group.

Pax_group_update : Function to update an entry in the table pax_group.

Pax_group_delete : Function to delete an entry fron the table pax_group.

Request_leg_add : Funtion to add an entry to the table request_legs. Entries such as request_id, leg_id, num_pax, orig_icao, dest_icao, e_dept_time, etc.are added.

Request_leg_update : Function to update an entry in the table request_legs.

Request_legs_delete : Function to delete an entry from the table request_legs.

Request_mission_add : Function to add an entry to the table request_mission .

Request_mission_update : Function to update an entry in the table request_mission.

Request_mission_delete : Function to delete an entry from the table request_mission.

Tail_add : Function to add an entry to the table tail.

Tail_update : Function to update an entry in the table tail.

Tail_delete : Function to delte an entry from the table tail.

SECTION 3

DATABASE SYSTEM MANUAL

The major part of the DESc user manual is devoted to the alphabetical listing of the API functions. It begins with a discussion of various data types supported in DESc. This is followed by a complete description of the database schema compiler. The API reference section covers type definitions, global constants, global variables, and environment variables in the system. Each and every API function is described with all the information needed to use them. Each function is listed on a separate page in alphabetical order.

Table 1 Data Types in DESc{tc "1 Data Types in DESc" \f t}

DESc TYPE	C TYPE
CHAR	char
STRING	N.A (char)
SHORT	short
USHORT	unsigned short
INT	int
UINT	unsigned int
LONG	long
ULONG	unsigned long
FLOAT	float
DOUBLE	double
DTULONG	N.A (unsigned long)
SERIAL	N.A (unsigned long)
TEXT	N.A (char)

I DATA TYPES

DESc supports 13 data types. This section describes these data types in detail. These data types and their C language equivalents are given in Table A.1. Four of these data types do not have an exact equivalent in C language. For these data types, their nearest C counterparts are given.

In the following sections, each of these data types is explained in detail. Storage requirements, range of values, and other related information are also discussed.

CHAR is used to store fixed length case sensitive character arrays. For this data type, the user should specify the length of the array. Strings are not null terminated. If you want to have them null terminated, then you should include one extra character in the length. However, for indexing purposes, a null character terminates the string. If there is no null character in the actual record, then the specified length is used. The size of a column of this type is given by the user. If no size is specified, then the default size, one, is assumed.

STRING is used to store fixed length case insensitive character arrays. This is similar to the previous type, but all comparisons will be case in-sensitive. In the actual record, values are stored with case sensitivity; but comparisons are case in-sensitive. This is useful for storing names, address, etc. The size of a column of this type is given by the user. If no size is specified, then the *default size*, one, is assumed.

SHORT, INT, & LONG are used for storing signed integers. **USHORT, UINT, & ULONG** are used for storing unsigned integers. **FLOAT & DOUBLE** are used for storing decimal values. The size and range for these data types are determined by the operating system and/or compiler. Table A.2 lists the size and the range for DOS & UNIX systems.

Table 2 Size and Range of Numeric Data Types{tc "2 Size and Range of Numeric Data Types" \f t}

TYPE	DOS		UNIX	
	S	RANGE	S	RANGE
SHORT	2	-32,768 to 32,767	2	-32,768 to 32,767
USHORT	2	0 to 65,535	2	0 to 65,535
INT	2	-32,768 to 32,767	4	-2,147,483,648 to 2,147,483,647
UINT	2	0 to 65,535	4	0 to 4,294,967,295
LONG	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
ULONG	4	0 to 4,294,967,295	4	0 to 4,294,967,295
FLOAT	4	$3.4 * 10^{-38}$ to $3.4 * 10^{38}$	4	$3.4 * 10^{-38}$ to $3.4 * 10^{38}$
DOUBLE	8	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$	8	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$

DTULONG is used for storing date, time, and date-time values. The date is stored as number of minutes elapsed since 01/01/01 12.00 AM. The internal storage format is an unsigned long integer, and, hence, requires four bytes. Any date or time till 31 December 8000, 11.59 PM can be stored in this column. Functions to convert values, to and from this format, and traditional date-time strings are available in the API.

SERIAL is used for generating unique integer values in a column. This is usually used for adding a unique field to a table. This data type is fully compatible with ULONG data type.

TEXT is used to store variable length case insensitive character arrays. This is the only variable length data type available in DESc. Any table with one or more columns of this type will have variable length records. A null character marks the end of the column while inserting records. This data type is usually used for storing remarks, addresses, etc.

II DATABASE SCHEMA COMPILER

The data definition language is implemented as a schema compiler in DESc. Database schema compiler is an important productivity tool in DESc. It provides an easy way for creating the database without writing any program to handle this task. This section describes the database schema compiler. First the format of the database schema source file is explained in detail. This discussion involves explanation of comments, commands, and identifiers used in the schema source file. This is followed by a discussion of the contents of the output files created by the schema compiler. The command line syntax and error message of the schema compiler are discussed last.

The schema compiler compiles the schema source file (ASCII text file) and creates the tables and the indexes in the database. It also creates a C source file and a header file to be used in the application program. If there are any errors in the schema source file, then they will be reported. Tables and indexes can also be created through API functions. But doing them with the schema compiler is the easiest way. Also, the source and header files created by the schema compilers are useful while calling other database API functions from the application program.

The schema source file is composed of comments and three types of commands. Comments are for documenting the schema source file. The three commands are (1) **DATABASE** command that gives the name of the database and memory configuration parameters, (2) **TABLE** command that describes the tables in the database, (3) **INDEX** command that describes the indexes in the database. The syntax of comments and these three commands are discussed below. In these discussions, the following conventions are used:

- Capitalized items should be typed as they are.
- Italicized items should be replaced by relevant names or types.
- Entries enclosed in square brackets are optional.

2.1 Comments

Any line with a semicolon (;) as the first non-blank character is considered to be a comment line. Comments are ignored by the schema compiler. Comments are used for documenting the schema source file.

2.2 Database Command

The syntax of the database command is as follows.

#DATABASE ***DbName*** **[*BufferCount* [*BufferSize* [*BlockSize*]]]**

The database command starts with the keyword #DATABASE. This is followed by the name of the database and optional memory configuration parameters on the same line. BufferCount gives the number of memory buffers used by the indexing scheme. BufferSize gives the size of each buffer. BlockSize gives the block size of index files in the disk. If there is more than one database command, then the last one takes effect. If there is no database command, then the schema source file name without any extension is taken as the database name. If any of the memory configuration parameters is omitted, default values are assumed for these parameters.

2.3 Table Command

The syntax of the table command is as follows.

#TABLE ***TblName***
ColumnDefn
...
#END

The table command starts with the keyword #TABLE. This is followed by the table name in the same line. This is followed by one or more lines defining each field in the table. This is followed by the keyword #END on its own line.

Syntax of the ColumnDefn is as follows.

ColName ***ColType*** **[*ColSize*]**

ColumnDefn starts with the name of the column. This is followed by the type of column that can be one of the thirteen types described in the previous section. You should give the complete name of the type. If the type is CHAR or STRING, then it may be followed by the size of the column. For other column types, size is not allowed.

2.4 Index Command

The syntax of the index command is as follows.

#INDEX ***IdxName*** ***TblName*** ***IdxType***
ColName
...

#END

The index command starts with the keyword #INDEX. This is followed by the name of the index. This is followed by the name of the table on which this index is defined. This is followed by the type of the index. Two types of indexes are allowed. One is UNIQUE, and the other one is DUPLICATE. All these entries should be typed on the same line. This is followed by one or more lines that list the column names that form this index. Each line lists one column name from the table specified in the first line. This will be followed by the keyword #END on its own line.

The schema compiler also recognizes one more command that allows the user to specify the referential integrity constraints in the database. But as referential integrity is not supported in the present version of the DESc, this command is left as an undocumented feature of the schema compiler.

2.5 Identifiers

All identifiers in the schema source file (DbName, TblName, IdxName, & ColName) should satisfy the following rules :

- Length should be less than or equal to 32.
- First character should be an alphabet.
- Other characters should be alphabets, digits (0-9), or underscore (_).
- Table names should be unique. In DOS version, the first 8 characters of the table names should be unique.
- First 28 characters of table names and index names should also be unique if you want to compile the header and source files created by schema compiler in an ANSI C compiler.
- The identifiers are not case sensitive.

These identifiers are used to form structures and global variables to be written in the output source and header files. These global names are formed by attaching certain prefixes to these identifiers.

2.6 Output Files

Schema compiler generates a C source file (extension .c) and a header file (extension .h). The name of the database is used for the name part of these files. In DOS, only the first eight characters of the database name is used. In the header file, structure types are defined for each table and index in the database. The header file also has external declarations for the global variables declared in the C source file. This header file should be included in the application program. The source file created by the schema compiler declares some initialized global variables for each table and index. This source file should be compiled and linked with other files in the application program. Identifiers for the type definitions and global variables are created by attaching some prefixes to the table names and index names. The purpose of these variables and the prefixes used are explained below.

The global variable DB_NAME is initialized with the name of the database. This could be used in any API function that needs the name of the database as one of the input parameters.

For each table, a structure type is defined. The prefix "TST_" is used for this type definition. Global variables are declared for each table using these defined types. These global variables are declared in the source file. The prefix "TGV_" is used for these global variable names. A character array is declared to store the name of the table. The prefix "TN_" is used for this array name. Four arrays are declared for each table. All these arrays will be of size n+1, where n is the number of columns in the table. The first array is an array of pointers. Each element in this array stores the address of one column in the global variable. The prefix "TCP_" is used for this array name. The second array is an array of strings. Each element in this array stores the name of one column in the table. The prefix "TCN_" is used for this array name. The third array is an array of integers. Each element in this array stores the data type of one column in the table. The prefix "TCT_" is used for this array name. The fourth array is also an array of integers. Each element in this array stores the length of one column in the table. The prefix "TCL_" is used for this array name.

For each index, a structure type is defined. The prefix "IST_" is used for this type definition. Global variables are declared for each index using these defined types. These global variables are declared in the source file. The prefix "IGV_" is used for these global variable names. A character array is declared to store the name of the index. The prefix "IN_" is used for this array name. An integer is declared to store the type of the index. The prefix "IT_" is used for this index type variable. Two arrays are declared for each index. All these arrays will be of size n+1, where n is the number of columns in that index. The first array is an array of pointers. Each element in this array stores the address of one column in the global variable. The prefix "ICP_" is used for this array name. The second array is an array of strings. Each element in this array stores the name of one column in the index. The prefix "ICN_" is used for this array name.

It is important to become familiar with these elements to use them in your application program. Using these variables instead of actual name strings should avoid run time errors

in your application as any typo in your application will be reported by the compiler. For example, any API function that requires the table name or the index name as one of the parameters could take the global variable (with prefix TN_ or IN_) defined in these files. The global variable with the prefix TGV_ and the global array of pointers with the prefix TCP_ could be used to pass parameters to the insert function.

2.7 Error Messages

The schema compiler displays three types of error messages. They are warnings, errors, and fatal errors. In case of warnings, the tables and output files are created. However, it is advisable to avoid warnings. In the case of errors, tables and output files are not created. The compiler proceeds with the compilation after recovering from the error condition. Of course, one error may lead to several other errors. Fatal errors terminate the compilation at that line. These are serious errors. The schema compiler displays the error messages along with the line number that caused the error condition. These error messages are displayed on the standard error, but could also be redirected to a file through a command line switch. There are 46 messages in all. Following are warnings, errors, and fatal errors displayed by the schema compiler. In the actual message, "XX" will be replaced by the appropriate string and "n" will be replaced by a numeric value. Type of error (warning, error, fatal error) will also be displayed in the actual message.

- USAGE : dbsc -[ce] filename
- Can't open input file "XX" for reading
- Can't open error file "XX". Writing to stderr
- Can't open file "XX" for writing. Exiting
- Compilation stopped with "n" error(s) & "n" Warning(s)
- Compilation success with "n" Warning(s)
- Creating the system tables
- Internal error. Bad recovery. Exiting
- Memory allocation error. Exiting
- Unexpected end of file
- Extra characters at the end of line "XX"
- Too many columns per index. Limit is "n"
- Keyword expected
- Keyword #END missing. Inserted
- Database name expected
- Database operating parameter expected
- Table name expected
- Index name expected
- Column name expected
- Index type expected
- Column type expected
- Keyword "XX" is invalid
- Keyword #END ignored
- Identifier "XX" too long. Extra characters ignored

- Identifier "XX" should start with a letter
- Identifier "XX" has invalid characters
- Non-numeric characters not allowed - "XX"
- Invalid index type - "XX"
- Invalid Column type - "XX"
- Table "XX" undefined
- Column "XX" undefined in table "XX"
- Primary and secondary table should be different
- Referring columns are of different type
- Referring columns are of different size
- No primary index on column "XX" in table "XX"
- No columns in the index "XX"
- Table "XX" redefined
- Column "XX" redefined in table "XX"
- Index "XX" redefined in table "XX"
- Column "XX" redefined in index "XX"
- Creation error in database "XX"
- Creation error in table "XX"
- Creation error in index "XX" in table "XX"
- First "n" chars of table/index name should be unique - "XX"
- First "n" chars of table name should be unique - "XX"
- Only one column can be of type SERIAL - "XX"

2.8 Command Line Syntax

The command line syntax for the schema compiler is given below:

dbsc [-ce] filename,

where **dbsc** stands for **database schema compiler**. The command line switch **-e** redirects the errors (if any) to the file with the same name as "filename" but with the extension "err". The switch **-c** tells the compiler to stop after the compilation. In this case, database and other output files are not created. The filename is the name of the file containing the schema source file. If there is no extension, then the default extension "sch" is appended to this filename.

Table 3 Type Definitions in Dbuser.h{tc "3 Type Definitions in Dbuser.h" \f t}

Type	C Type	Purpose
CHAR	char	for the data type CHAR
STRING	char	for the data type STRING
SHORT	short	for the data type SHORT
USHORT	unsigned short	for the data type USHORT
INT	int	for the data type INT
UINT	unsigned int	for the data type UINT
LONG	long	for the data type LONG
ULONG	unsigned long	for the data type ULONG
FLOAT	float	for the data type FLOAT
DOUBLE	double	for the data type DOUBLE
DTULONG	unsigned long	for the data type DTULONG
SERIAL	unsigned long	for the data type SERIAL
TEXT	char	for the data type TEXT
DBDT	structure	for date time structure format

III. API REFERENCE

In this section, the API functions available in DESc are described in detail. To call these functions, you have to include the header file **dbuser.h**. Before explaining various functions, type definitions, global variables, global constants, and environment variables associated with DESc are described.

3.1 Type Definitions

There are 14 type definitions in the file **dbuser.h**. Thirteen of these are for the data types in DESc. The last one is a structure for supporting an alternative data-time format. These types, their equivalent C types, and their purpose are listed in Table A.3.

The DBDT structure is given below :

```
typedef     struct
            {
            int     Day;
            int     Month;
            int     Year;
            int     Hour;
            int     Minute;
            } DBDT;
```

3.2 Global Constants

Global constants are defined to help in passing or returning parameters to and from API functions. Each of these constants and their purpose is given below :

- IT_UNIQUE — unique index
- IT_DUPLICATE — duplicate index
- CT_CHAR — case sensitive character array
- CT_STRING — case in-sensitive character array
- CT_SHORT — short integer
- CT_USHORT — unsigned short integer
- CT_INT — integer
- CT_UINT — unsigned integer
- CT_LONG — long integer
- CT_ULONG — unsigned long integer
- CT_FLOAT — small decimal value
- CT_DOUBLE — large decimal value
- CT_DTULONG — date & time value
- CT_SERIAL — system supplied sequential record number
- CT_TEXT — variable length character array

- **QT_SELECT** — **select operation**
- **QT_UPDATE** — **update operation**
- **QT_DELETE** — **delete operation**
- **QT_SELTMP** — **select into temporary table**
- **CD_UNIQUE** — **only distinct values allowed**
- **CD_DUPLICATE** — **duplicate values allowed**
- **SP_NOSORT** — **don't need to sort**
- **OP_EQ** — **x == Value1**
- **OP_NE** — **x != Value1**
- **OP_GT** — **x > Value1**
- **OP_GE** — **x >= Value1**
- **OP_LT** — **x < Value1**
- **OP_LE** — **x <= Value1**
- **OP_IR** — **x >= Value1 AND x <= Value2**
- **OP_NR** — **x < Value1 OR X > Value2**

3.3 Global Variables

There are only three global variables in **dbuser.h**. The first global variable (char *DbPhyIdxName) gives the name of the physical index. The second global variable (char *DbQryIdxName) gives the name of the index created to retrieve the records in the requested (sorted) order. The last global variable (int DbError) gives the error number returned by various API *functions*. There are 47 constants defined in the **dbuser.h** for different error numbers returned in this global variable.

3.4 Environment Variables

DESc expects two environment variables to be set to handle the path for the database operations. These environment variables are DBDIR and DBTMP. DBDIR gives the complete path of the database sub-directory. Every database is stored in a separate directory, with the same name as the database, under the directory specified in this environment variable. DBTMP is the temporary directory where the database creates all temporary files on the fly. It is important that these environment variables are set with proper values to avoid any problems in database operations.

3.5 Functions By Category

API functions are available under five major categories. The categories and the functions in each of these categories are listed below :

Data Definition Functions

- DbCreate
- DbDrop
- DbCreateTable
- DbDropTable
- DbCreateIndex
- DbDropIndex

Data Manipulation Functions

- DbOpenQuery
- DbCloseQuery
- DbVerifyQuery
- DbSpecifyTable
- DbSpecifyColumn
- DbSpecifyAllColumns
- DbSpecifyJoin
- DbSpecifySelection
- DbSpecifyAlias
- DbOpenCursor
- DbCloseCursor
- DbReadCursor
- DbUpdateCursor
- DbPerformDelete
- DbPerformSelTmp
- DbInsertRecord

Database Utility Functions

- DbInit
- DbExit
- DbFlush
- DbGetErrorMsg
- DbPrintErrorMsg
- DbDumpTable
- DbEmptyTable
- DbLoadTable

Schema Information Functions

- DbDumpDbInfo
- DbDumpIndexInfo
- DbDumpTableInfo
- DbGetDbInfo
- DbGetTableInfo
- DbGetTableColumnInfo
- DbGetIndexInfo
- DbGetIndexColumnInfo

General Purpose Functions

- DbBuildDateTimeString
- DbParseDateTimeString
- DbConvertToLongDT
- DbConvertToStructDT
- DbValidateDT
- DbGetDayOfWeek
- DbCheckNull
- DbSetNull

3.6 Sample API Look-up Entry

Every function that is available to the user is explained here. The following template gives you an idea of how to use this section. All database API function names start with the prefix Db. The rest of the name will be (at least in most of the cases) a verb followed by a noun.

FunctionName

Function **Summary of what the function does**

Syntax **#include <header.h>**
(The header file containing the prototype for function or definitions of constants, enumerated types, etc., used by the **function**)

function (modifier parameter[,...])
(The declaration syntax for the **function**; parameter names are italicized. The [,...] indicates that more parameters and their modifiers may follow.

Remarks This section describes what the **function** does, the parameters it takes, and any details you need to use the **function**.

Return value The value that **function** return (if any) is given here. If function sets any global variables, their values are also listed.

See also Other API functions related to this **function** are listed here. A function name that contains ellipsis (...) indicates a family of functions.

Example sample code showing the use of **function**

DbBuildDateTimeString

Function **Builds a date time string from the DBDT structure**

Syntax
```
#include <dbuser.h>
int DbBuildDateTimeString (DBDT *StructDT, char
StringDT);
```

Remarks This function builds a string in StringDT from the values in the date time structure StructDT. The string will be of the form "yyyy-mm-dd hh:mm" if the value represents both date and time; "mm/dd/yyyy" if the value represents only a date; "hh:mm" if the value represents only a time.

Return value This function returns OK on success. If the values in the DBDT structure are invalid, then this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbConvertToLongDT
DbConvertToStructDT
DbGetDayOfWeek
DbParseDateTimeString
DbValidateDT

Example Refer the sample source files dembench.c and demgsbrs.c

DbCheckNull

Function **Check if the value in a column is NULL**

Syntax

```
#include <dbuser.h>
int DbCheckNull (int Type, void *Value);
```

Remarks This function checks if the value stored in the void pointer is a NULL value for the specified column Type. Type should be one of the valid column types. Value should point to a variable of relevant type. Otherwise behavior is undefined. This function is used to check for null values in columns after retrieving a record from the database.

Return value This function returns YES if the Value is NULL. Otherwise this function returns NO. In case of an error, error number is returned in the global variable DbError.

See also DbSetNull

Example Refer the sample source files dembench.c and demgsbrs.c

DbCloseCursor

Function **Closes a cursor opened by DbOpenCursor**

Syntax
```
#include <dbuser.h>
int DbCloseCursor (QUERY *Qry);
```

Remarks This function closes a cursor opened by DbOpenCursor. Qry should be a valid query returned by a DbOpenQuery.

Return value This function returns OK if the cursor is closed. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCloseQuery
DbOpenCursor
DbOpenQuery
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbCloseQuery

Function **Closes a query opened by DbOpenQuery**

Syntax
```
#include <dbuser.h>
int DbCloseQuery ( QUERY Qry);
```

Remarks This function closes a query opened by DbOpenQuery. Qry should be a valid query returned by a DbOpenQuery.

Return value This function returns OK if the query is closed. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCloseCursor
DbOpenCursor
DbOpenQuery
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbConvertToLongDT

Function **Convert a date in structure form to a date in long form**

Syntax
```
#include <dbuser.h>
int DbConvertToLongDT (DBDT *StructDT, DTULONG
*LongDT);
```

Remarks This function converts a date given as a DBDT structure into the unsigned long integer format. This is the internal format needed for column values.

Return value This function returns OK on success. If the values in the DBDT structure are invalid, then this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbBuildDateTimeString
DbConvertToStructDT
DbGetDayOfWeek
DbParseDateTimeString
DbValidateDT

Example Refer the sample source files dembench.c and demgsbrs.c

DbConvertToStructDT

Function **Convert a date in long form to a date in structure form**

Syntax

```
#include <dbuser.h>
int DbConvertToStructDT (DTULONG LongDT, DBDT
*StructDT);
```

Remarks This function converts a date given as an unsigned long integer into a date in DBDT structure form.

Return value This function returns OK on success. If the date is invalid, then this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbBuildDateTimeString
DbConvertToLongDT
DbGetDayOfWeek
DbParseDateTimeString
DbValidateDT

Example Refer the sample source files dembench.c and demgsbrs.c

DbCreate

Function **Creates a new database**

Syntax
```
#include <dbuser.h>
int DbCreate (char *DbName, int BufferCount, int
BufferSize, int BlockSize);
```

Remarks This function creates a new database with the name DbName. It is important to have a sub-directory named DbName under the directory declared in the environment variable DBDIR. This directory should be empty. BufferCount and BufferSize give the count and size of the memory buffers to be used by the indexing scheme. BlockSize gives the size of the index file blocks in the disk. If any of these three parameters are zero or too small, then default values are silently used.

Return value This function returns OK if it succeeds in creating the database. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbDrop
DbDumpDbInfo
DbFlush
DbGetDbInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbCreateIndex

Function **Creates a new index on an existing table**

Syntax

```
#include <dbuser.h>
int DbCreateIndex (char *TblName, char *IdxName, int
IndexType, char *ColNames[ ]);
```

Remarks This function creates a new index. The name of the new index is given in IdxName. TblName should be a valid existing table in the current database. IndexType gives the type of index. It can be either IT_UNIQUE or IT_DUPLICATE. ColNames is an argv style array of pointers to column names that constitute the index.

Return value This function returns OK if it succeeds in creating the index. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbDropIndex
DbDumpIndexInfo
DbGetIndexColumnInfo
DbGetIndexInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbCreateTable

Function **Creates a new table in the current database**

Syntax

```
#include <dbuser.h>
int DbCreateTable (char *TblName, char *ColNames[ ], int
ColTypes[ ], int ColLengths[ ]);
```

Remarks This function creates a new table. The name of the new table is given in TblName. ColNames is an argv style array of pointers to column names that constitute the table. ColTypes is an array of integers giving the type of each column in the new table. ColLengths is an array of integers giving the length of each column. Entries in this last array are optional.

Return value This function returns OK if it succeeds in creating the table. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbDropTable
DbDumpTable
DbDumpTableInfo
DbEmptyTable
DbLoadTable
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbDrop

Function **Drops an existing database**

Syntax
```
#include <dbuser.h>
int DbDrop (char *DbName);
```

Remarks This function drops an existing database with the name DbName. All tables and indexes in the database will be lost forever. Current database cannot be destroyed.

Return value This function returns OK if it succeeds in dropping the database. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCreate
DbDumpDbInfo
DbFlush
DbGetDbInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbDropIndex

Function **Drops an existing index from a table**

Syntax
```
#include <dbuser.h>
int DbDropIndex (char *TblName, char *IdxName);
```

Remarks This function drops the index IdxName from the table TblName. TblName should be a valid table in the current database, and IdxName should be a valid index in this database. Other indexes on this table are not disturbed.

Return value This function returns OK if it succeeds in dropping the index. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCreateIndex
DbDumpIndexInfo
DbGetIndexColumnInfo
DbGetIndexInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbDropTable

Function **Drops an existing table in the current database**

Syntax
```
#include <dbuser.h>
int DbDropTable (char *TblName);
```

Remarks This function drops an existing table from the current database. TblName should be a valid table in the current database. All records and indexes are lost forever.

Return value This function returns OK if it succeeds in dropping the table. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCreateTable
DbDumpTable
DbDumpTableInfo
DbEmptyTable
DbLoadTable
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbDumpDbInfo

Function **Displays information about current database**

Syntax
```
#include <dbuser.h>
int DbDumpDbInfo ( );
```

Remarks This function displays information about the current database on the standard output. This information includes all information about the tables and indexes on the database. This function is used mostly as a debug function to dump the schema information during initial development.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbDumpIndexInfo
DbDumpTableInfo
DbGetDbInfo
DbGetIndexColumnInfo
DbGetIndexInfo
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbDumpIndexInfo

Function **Displays information about an index**

Syntax
```
#include <dbuser.h>
int DbDumpIndexInfo ( char *TblName, char *IdxName);
```

Remarks This function displays information about an index on the standard output. This information includes index type, number of columns, names and types of these columns, among other things. TblName and IdxName identify the index in the database. This function is used mostly as a debug function to dump the schema information during initial development.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbDumpDbInfo
DbDumpTableInfo
DbGetDbInfo
DbGetIndexColumnInfo
DbGetIndexInfo
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbDumpTable

Function **Dumps all records in a table into a file**

Syntax
```
#include <dbuser.h>
int DbDumpTable (char *TblName, char IdxName, char
*DumpFileName, char *Separator);
```

Remarks This function dumps all records in a table into a file. TblName is the table to be dumped. IdxName is the order for retrieving records from the table. NULL in this parameter allows the records to be retrieved in physical sequence. DumpFileName is the full path name of the file where records will be written as ASCII text. Records will be re-directed to standard output if this parameter is NULL. Separator gives the column separator in the output file. Default separator is pipe symbol (|) and will be in effect if this parameter is NULL. Records are written one per line.

Return value This function returns OK if it succeeds in dumping the table. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbDumpTable
DbEmptyTable
DbLoadTable

Example Refer the sample source files dembench.c and demgsbrs.c

DbDumpTableInfo

Function **Displays information about a table**

Syntax

```
#include <dbuser.h>
int DbDumpTableInfo ( char *TblName);
```

Remarks This function displays information about a table on the standard output. This information includes table type, number of columns, names and types of these columns among other things. TblName identifies the table name in the database. This function is used mostly as a debug function to dump the schema information during initial development.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbDumpDbInfo
DbDumpIndexInfo
DbGetDbInfo
DbGetIndexColumnInfo
DbGetIndexInfo
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbEmptyTable

Function **Deletes all records in a table**

Syntax
```
#include <dbuser.h>
int DbEmptyTable (char *TblName);
```

Remarks This function deletes all records in a table. The table or the indexes are not touched. TblName is the table to be emptied.

Return value This function returns OK if it succeeds in emptying the table. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCreateTable
DbDropTable
DbDumpTable
DbLoadTable

Example Refer the sample source files dembench.c and demgsbrs.c

DbExit

Function **Closes database processing**

Syntax
```
#include <dbuser.h>
int DbExit ( );
```

Remarks This function closes database processing. All tables and indexes are written to the disk. DbInit should have been called before calling this. DbInit automatically sets function pointers to call this function at exit.

Return value This function returns OK if it succeeds in closing the database processing. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also DbFlush
DbInit

Example Refer the sample source files dembench.c and demgsbrs.c

DbFlush

Function **Flushes all database changes to disk**

Syntax
```
#include <dbuser.h>
int DbFlush ( );
```

Remarks This function flushes all changes in the database to disk. This function should be called if you do not want to lose your changes. Make it a point to call this function once in a while to avoid any surprises.

Return value This function returns OK if it succeeds in flushing all database changes disk. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also DbExit
DbInit

Example Refer the sample source files dembench.c and demgsbrs.c

DbGetDayOfWeek

Function **Returns the day of week for a date in long form**

Syntax
```
#include <dbuser.h>
int DbGetDayOfWeek (DTULONG LongDT);
```

Remarks This function returns the day of week for a date given in the unsigned long integer format. The day of week is returned as an integer in the range 0 to 6.

Return value Return the day of week as an integer in the range 0 - 6. 0-Sun, 1-Mon, 2-Tue, 3-Wed, 4-Thu, 5-Fri, 6-Sat.

See also
DbBuildDateTimeString
DbConvertToLongDT
DbConvertToStructDT
DbParseDateTimeString
DbValidateDT

Example Refer the sample source files dembench.c and demgsbrs.c

DbGetDbInfo

Function **Returns information about current database**

Syntax

```
#include <dbuser.h>
int DbGetDbInfo ( char **DbName, int *TblCount, char
***TblNames);
```

Remarks This function returns information about the current database. Name of the database, number of tables in the database, names of the tables are the three pieces of information returned in the three parameters. This function is used mostly as an inquiry function to know the schema information.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbDumpDbInfo
DbDumpIndexInfo
DbDumpTableInfo
DbGetIndexColumnInfo
DbGetIndexInfo
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbGetErrorMsg

Function **Return pointer to detailed error message.**

Syntax

```
#include <dbuser.h>
char *DbGetErrorMsg (int ErrorNbr);
```

Remarks This function returns pointer to detailed error message for an error number. ErrorNbr is the value stored in the global variable DbError during an error condition. Constants defined in dbuser.h for errors could also be used for passing values in ErrorNbr.

Return value Returns pointer to the error message. If the error number is invalid, then pointer to a default message (no errors) is returned.

See also DbPrintErrorMsg

Example Refer the sample source files dembench.c and demgsbrs.c

DbGetIndexColumnInfo

Function **Returns information about a column in an index**

Syntax

```
#include <dbuser.h>
int DbGetIndexColumnInfo ( char *TblName, char *IdxName,
char *ColName, int *ColPosn, int *ColType, int *ColLength);
```

Remarks This function returns information about a column in an index. TblName, IdxName, and ColName identifies a column in an index. Position of this column in this index, type of the column, and the length of the column are the information returned in the next three parameters. This function is used mostly as an inquiry function to know the schema information.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbDumpDbInfo
DbDumpIndexInfo
DbDumpTableInfo
DbGetDbInfo
DbGetIndexInfo
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbGetIndexInfo

Function **Returns information about an index**

Syntax
```
#include <dbuser.h>
int DbGetIndexInfo ( char *TblName, char *IdxName, int
*IdxType, int *ColCount, char ***ColNames);
```

Remarks This function returns information about an index. TblName and IdxName identifies the index in the database. Type of index, number of columns, and names of columns are the information returned in the next three parameters. This function is used mostly as an inquiry function to know the schema information.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbDumpDbInfo
DbDumpIndexInfo
DbDumpTableInfo
DbGetDbInfo
DbGetIndexColumnInfo
DbGetTableColumnInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbGetTableColumnInfo

Function **Returns information about a column in a table**

Syntax

```
#include <dbuser.h>
int DbGetTableColumnInfo ( char *TblName,  char
*ColName, int *ColPosn, int *ColType, int *ColLength);
```

Remarks This function returns information about a column in a table. TblName and ColName identify a column in a table. Position of this column in this table, type of the column, and the length of the column are the information returned in the next three parameters. This function is used mostly as an inquiry function to know the schema information.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbDumpDbInfo
DbDumpIndexInfo
DbDumpTableInfo
DbGetDbInfo
DbGetIndexColumnInfo
DbGetIndexInfo
DbGetTableInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbGetTableInfo

Function **Returns information about a table**

Syntax

```
#include <dbuser.h>
int DbGetTableInfo ( char *TblName, int *TblType, int
*ColCount, char ***ColNames, int *IdxCount, char
***IdxNames);
```

Remarks This function returns information about a table. TblName identifies the table name in the database. Table type, number of columns, column names, index count, and index names are the information returned in the next five parameters. This function is used mostly as an inquiry function to know the schema information.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbDumpDbInfo
DbDumpIndexInfo
DbDumpTableInfo
DbGetDbInfo
DbGetIndexColumnInfo
DbGetIndexInfo
DbGetTableColumnInfo

Example Refer the sample source files dembench.c and demgsbrs.c

DbInit

Function **Initializes database processing**

Syntax

```
#include <dbuser.h>
int DbInit ( char *DbName, int BufferCount);
```

Remarks This function initializes database processing. DbName is an existing database to be initialized for processing. BufferCount is the number of buffers to be used for the indexing scheme during processing. If this parameter is zero, then BufferCount given during DbCreate is silently used. DbInit should be the first function to be called in an application program. No other function (DbCreate and DbDrop are the only exceptions) should be called before calling DbInit.

Return value This function returns OK if it succeeds in initializing the database processing. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also DbExit
DbFlush

Example Refer the sample source files dembench.c and demgsbrs.c

DbInsertRecord

Function **Inserts a new record into a table**

Syntax
```
#include <dbuser.h>
int DbInsertRecord (char *TblName, char *Columns[ ]);
```

Remarks This function inserts a record in the table named TblName. Columns is an argv style array of pointers to the column values in the record. The global variable TGV_xxxx could be used to fill in the values, and the array of pointers TCP_xxxx could be used for the second parameter. These variables are declared in the output files generated by the schema compiler. Here xxxx should be replaced by the table name as typed in the schema source file. If there are any unique indexes created in the table, then duplicate values are not allowed in those columns.

Return value This function returns OK if the record is inserted into the table. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbPerformDelete
DbPerformSelTmp
DbReadCursor
DbUpdateCursor

Example Refer the sample source files dembench.c and demgsbrs.c

DbLoadTable

Function **Loads records from a text file into a table**

Syntax
```
#include <dbuser.h>
int DbLoadTable (char *TblName, char *LoadFileName);
```

Remarks This function loads records from an ASCII text file into a table. TblName is the table to be loaded. LoadFileName is the complete path of the text file. The file should have records one per line with columns separated by pipe symbol (|). A null value is identified by two successive pipe symbols.

Return value This function returns OK if it succeeds in loading the table. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCreateTable
DbDropTable
DbDumpTable
DbEmptyTable

Example Refer the sample source files dembench.c and demgsbrs.c

DbParseDateTimeString

Function **Parses a string to build a date-time in DBDT structure form**

Syntax

```
#include <dbuser.h>
int DbParseDateTimeString (char *StringDT, DBDT
*StructDT);
```

Remarks This function parses a string and builds a date-time value in DBDT structure form in StructDT. StringDT is a string in one of the following forms. "yyyy-mm-dd hh:mm" for date-time values; "mm/dd/yyyy" for date values; "hh:mm" for time values.

Return value This function returns OK on success. If the string is not in one of the three valid formats, then this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbBuildDateTimeString
DbConvertToLongDT
DbConvertToStructDT
DbGetDayOfWeek
DbValidateDT

Example Refer the sample source files dembench.c and demgsbrs.c

DbPerformDelete

Function **Performs the SQL delete operation**

Syntax
```
#include <dbuser.h>
int DbPerformDelete (QUERY *Qry);
```

Remarks This function performs the delete operation specified in the query structure. Qry should be a valid query returned by DbOpenQuery function. Qry should be of type QT_DELETE and should have been verified using DbVerifyQuery. The specified SQL is performed and all matching records are deleted from the table specified in DbSpecifyTable. All deleted records are lost forever.

Return value This function returns OK if the delete operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCloseQuery
DbPerform. . .
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbPerformSelTmp

Function **Selects records into a temporary table**

Syntax
```
#include <dbuser.h>
int DbOpenCursor (QUERY *Qry);
```

Remarks This function selects records into a temporary table. Qry should be a valid query returned by DbOpenQuery function. Qry should be of type QT_SELTMP and should have been verified using DbVerifyQuery. The specified SQL is performed and all matching records are inserted into the temporary table specified in DbSpecifyTable. Any sorting order specified is available through the index defined in the global variable DbQryIdxName.

Return value This function returns OK if the select operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCloseQuery
DbPerform. . .
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbPrintErrorMsg

Function **Prints error message on stdout**

Syntax
```
#include <dbuser.h>
char *DbPrintErrorMsg ( );
```

Remarks This function prints the error message for the error number in global variable DbError. The message is displayed in the standard output.

Return value Returns pointer to the error message. If the DbError is invalid, then a default message (no errors) is displayed .

See also DbGetErrorMsg

Example Refer the sample source files dembench.c and demgsbrs.c

DbOpenCursor

Function **Opens an SQL cursor for select / update operations**

Syntax
```
#include <dbuser.h>
int DbOpenCursor (QUERY *Qry, int *RecCount);
```

Remarks This function opens an SQL cursor for select and delete operations. Qry should be a valid query returned by DbOpenQuery function. Qry should be of type QT_SELECT or QT_UPDATE and should have been verified using DbVerifyQuery. The specified SQL is performed and all matching records are inserted into a temporary system table for further processing by DbReadCursor and DbUpdateCursor. This function returns the number of matching records in the second parameter RecCount.

Return value This function returns OK if the SQL cursor open operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbCloseCursor
DbCloseQuery
DbPerform. . .
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbOpenQuery

Function **Opens a query structure for SQL processing**

Syntax

```
#include <dbuser.h>
QUERY *DbOpenQuery (int QryType);
```

Remarks This function opens a query structure for SQL processing. The pointer returned to a new query structure should be used in other functions. QryType is one of the four query types allowed in the system. You could use the following constants defined in the header file for this purpose. QT_SELECT, QT_SELTMP, QT_DELETE, QT_UPDATE.

Return value This function returns OK if the query open operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbCloseQuery
DbPerform. . .
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbReadCursor

Function **Reads the next record from the SQL cursor**

Syntax
```
#include <dbuser.h>
int DbReadCursor (QUERY *Qry);
```

Remarks This function reads the next record from the SQL cursor. Qry should be a valid query returned by DbOpenQuery function. Qry should be of type QT_SELECT or QT_UPDATE and should have been verified using DbVerifyQuery. You should call DbOpenCursor before calling this function. The retrieved values are stored in the location specified in DbSpecifyColumn or DbSpecifyTable, depending upon the type of the query.

Return value This function returns OK if the read cursor operation is successful. If there are no records to read, then this function returns EOI. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbPerform. . .
DbOpenCursor
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbSetNull

Function **Sets a variable to NULL value**

Syntax
```
#include <dbuser.h>
int DbSetNull (int Type, void *Value);
```

Remarks This function sets NULL to the location pointed by the pointer Value. Type gives the data type of this location. This function is used to set null values for column values while inserting a record.

Return value This function returns YES if Value is set to NULL. Otherwise this function returns NO. In case of an error, error number is returned in the global variable DbError.

See also DbSetNull

Example Refer the sample source files dembench.c and demgsbrs.c

DbSpecifyAlias

Function **Specifies an alias for a table to help self joins**

Syntax

```
#include <dbuser.h>
int DbSpecifyAlias (QUERY *Qry, char *TblName, char
*AliasName);
```

Remarks This function specifies an alias name for a table. This function will be used in queries involving self joins. The AliasName could be used in the query from this point as another table in the database. AliasName should not conflict with other table names or alias names. Qry should be a valid query returned by DbOpenQuery function.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbSpecifyAllColumns

Function **Specifies a set of columns to be retrieved in the result**

Syntax

```
#include <dbuser.h>
int DbSpecifyAllColumns (QUERY *Qry, char *TblName, char
*Columns[ ]);
```

Remarks This function specifies a set of columns to be retrieved in the result. This is a short cut to retrieve all columns in a table in the query output. The individual columns in this table should not be used by DbSpecifyColumn either before or after this function. Qry should be a valid query returned by DbOpenQuery function. Qry should be of type QT_SELECT. The third parameter specifies the location for storing the column values by DbReadCursor. You cannot specify distinct clause and sorting order for the columns using this function.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbSpecifyColumn

Function **Specifies a column to be retrieved in the result**

Syntax

```
#include <dbuser.h>
int DbSpecifyColumn (QUERY *Qry, char *TblName, char
*ColName, void *HostName, int Distinct, int SortPosn, char
*NewName);
```

Remarks This function specifies a column to be retrieved in the result. TblName and ColName parameters specify the column to be retrieved. Qry should be a valid query returned by DbOpenQuery function. Qry should be of type QT_SELECT or QT_SELTMP. The fourth parameter HostName specifies the location for storing the column value by DbReadCursor. Distinct parameter specifies if this column can contain duplicate values. CD_UNIQUE and CD_DISTINCT are two constants defined for this purpose. SortPosn specifies the sorting order for this column. You can use the constant SP_NOSORT not to sort this column. NewName could be used to specify a new name for this column if the query is of type QT_SELTMP. This is just to give different names to columns that come from different tables with the same name.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbSpecifyJoin

Function **Specifies a join condition**

Syntax

```
#include <dbuser.h>
int DbSpecifyJoin (QUERY *Qry, char *PriTblName, char
*PriColName, char *SecTblName, char *SecColName, int
Operator);
```

Remarks This function specifies a join condition. PriTblName and PriColName identify one of the columns to be joined. PriTblName and PriColName identify the other column to be joined. Operator specifies the comparison condition. You can use one of the six traditional relational operators for this. The constants OP_EQ, OP_NE, OP_GT, OP_GE, OP_LT, and OP_LE are defined for this purpose. Qry should be a valid query returned by DbOpenQuery function.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbSpecifySelection

Function **Specifies a selection condition**

Syntax
```
#include <dbuser.h>
int DbSpecifySelection (QUERY *Qry, char *TblName, char
*ColName, void *Value1, void *Value2, int Operator);
```

Remarks This function specifies a selection condition. TblName and ColName identify the column on which selection condition is to be applied. Value1 and Value2 are the values to be used in comparison. Value2 can be NULL. Operator specifies the comparison condition. You can use one of the six traditional relational operators if Value2 is NULL. The constants OP_EQ, OP_NE, OP_GT, OP_GE, OP_LT, and OP_LE are defined for this purpose. If Value2 is not NULL, then two special operators OP_IR and OP_NR should be used. OP_IR retrieves all records in the range of the two values (both inclusive). OP_NR retrieves all records not in the range. Qry should be a valid query returned by DbOpenQuery function.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbSpecifyTable

Function **Specifies a table for Delete, Update, and SelTmp**

Syntax

```
#include <dbuser.h>
int DbSpecifyTable (QUERY *Qry, char *TblName, char
*OldRecord[ ], char *NewRecord[ ]);
```

Remarks This function specifies a table for Delete, Update, and SelTmp operations. TblName specifies the table name. For a delete query, this is the table from which records are to be deleted. For a update query, this is the table in which records are to be updated. For a SelTmp query, this is the name of the temporary table. The next two parameters are used only in an update query. The first set of pointers, OldRecord, are filled in by the DbReadCursor function. The second set of parameters are also filled by DbReadCursor, but in addition to that, these pointers are used by DbUpdateCursor to get the new column values. So these locations will be updated by the application program between DbReadCursor and DbUpdateCursor. Qry should be a valid query returned by DbOpenQuery function. Qry should be one of the three types namely, QT_SELTMP, QT_DELETE, and QT_UPDATE.

Return value This function returns OK if the operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbOpenQuery
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbUpdateCursor

Function **Updates a record in the table**

Syntax

```
#include <dbuser.h>
int DbUpdateCursor (QUERY *Qry);
```

Remarks This function updates a record using the SQL cursor. Qry should be a valid query returned by DbOpenQuery function. Qry should be of type QT_UPDATE and should have been verified using DbVerifyQuery. You should call DbReadCursor before calling this function. DbReadCursor fills in OldRecord and NewRecord pointers specified in DbSpecifyTable with a selected record from the table. DbUpdateCursor searches for the record as in OldRecord and updates it with the values in NewRecord. It is very important not to disturb the OldRecord between calling DbReadCursor and DbUpdateCursor. The changes to be made into the record should be done using the copy of the record in NewRecord. Checks like duplicate values on unique indexes are performed before update.

Return value This function returns OK if the update cursor operation is successful. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also

DbOpenCursor
DbOpenQuery
DbReadCursor
DbSpecify. . .
DbVerifyQuery

Example Refer the sample source files dembench.c and demgsbrs.c

DbValidateDT

Function **Validates a date-time given in structure form**

Syntax
```
#include <dbuser.h>
int DbValidateDT (DBDT *StructDT);
```

Remarks This function validates a date given as a DBDT structure. It checks for leap years and range of values for different elements in the structure.

Return value This function returns OK on if the date-time structure is valid. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbBuildDateTimeString
DbConvertToLongDT
DbConvertToStructDT
DbGetDayOfWeek
DbParseDateTimeString

Example Refer the sample source files dembench.c and demgsbrs.c

DbVerifyQuery

Function **Verifies a query structure**

Syntax
```
#include <dbuser.h>
int DbVerifyQuery (QUERY *Qry);
```

Remarks This function verifies a query structure. Qry should be a valid query returned by DbOpenQuery function. This function validates various parameters specified by DbSpecify. . . functions. A query should be verified successfully before it could be performed. This function basically checks to verify if all necessary parameters have been specified and there are no conflicts.

Return value This function returns OK if the query satisfies all requirements. Otherwise this function returns ERROR. In case of an error, error number is returned in the global variable DbError.

See also
DbPerform. . .
DbOpenQuery
DbSpecify. . .

Example Refer the sample source files dembench.c and demgsbrs.c